2010-08-04

A Hybrid System for Glossary Generation of Feature Film Content for Language Learning

Ryan Arthur Corradini
Brigham Young University - Provo

A Hybrid System for Glossary Generation of Feature Film Content for Language Learning

Ryan A. Corradini

A project submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Arts

Alan K. Melby, Chair
Dirk Elzinga
Michael Bush

Department of Linguistics and English Language

Brigham Young University

December 2010

ABSTRACT

A Hybrid System for Glossary Generation of Feature

Film Content for Language Learning

Ryan A. Corradini

Department of Linguistics and English Language

Master of Arts

This report introduces a suite of command-line tools created to assist content developers with the creation of rich supplementary material to use in conjunction with feature films and other video assets in language teaching. The tools are intended to leverage open-source corpora and software (the OPUS OpenSubs corpus and the Moses statistical machine translation system, respectively), but are written in a modular fashion so that other resources could be leveraged in their place.

The completed tool suite facilitates three main tasks, which together constitute this project. First, several scripts created for use in preparing linguistic data for the system are discussed. Next, a set of scripts are described that together leverage the strengths of both terminology management and statistical machine translation to provide candidate translation entries for terms of interest. Finally, a tool chain and methodology are given for enriching the terminological data store based on the output of the machine translation process, thereby enabling greater accuracy and efficiency with each subsequent application.

Keywords: electronic film review, language instruction, statistical machine translation, terminology management

ACKNOWLEDGEMENTS

I would like to thank Dr. Alan Melby for his mentorship and friendship over the years, particularly in the past several years as we have worked together to shape this project into its final form. His insights and advice have been invaluable. Thanks also go to Drs. Mike Bush and Deryle Lonsdale for their support and feedback as the project evolved from its initial form, and to Dr. Dirk Elzinga who graciously stepped in at the last minute to take Dr. Lonsdale's place when he was away leading a study abroad program. Finally, I owe my wife a debt of gratitude for the many evenings she spent alone while I toiled into the wee hours of the night. Without her love, encouragement, and patience with my eccentricities, this project would never have been completed successfully. Thank you, Mary.

TABLE OF CONTENTS

CHAPTER 1: BACKGROUND & JUSTIFICATION

1.1. DIFFERING METHODOLOGIES. It could well be said that for as long as there have been teachers, there have been differences of opinion about the best way to teach. While such a sweeping generalization may be difficult to prove, at least within the context of second-language instruction, it certainly seems apt. Some educators have taken a grammar-based approach, teaching language from the bottom up. Others insist that an immersive approach is best, fully surrounding students in their target language. Further, a great many variations exist between these two extremes (Mora 1998). The Electronic Film Review (hereafter EFR) project takes something of a hybrid approach, juxtaposing authentic natural language content (in the form of commercial films on DVD) with annotative material that allows the student to supplement their understanding of the target language when and how they choose (Melby, 2004). It is the intent of this project to build upon the work done by Melby and others, to provide tools that will enable content developers for language instruction to more easily create rich, immersive natural language experiences for their students.

1.2. NATURAL LANGUAGE EXPOSURE THROUGH VIDEO. For over 30 years, language educators have recognized the potential benefit of using video to teach language in a real-world context, but have often struggled to fit it into an already-crowded curriculum. Researchers have found that students are able to learn language skills from multimedia presentations, but are often quick to point out that such content must be supplemented by the instructor in order to be properly contextualized and understood (Secules, Herron and Tomasello 1992).

A second trend in the past decade has been notable growth in the ubiquity of personal computers in the classroom, as technology has continued to grow in sophistication even as it has decreased in cost. With the advent of first the CD-ROM, and later the DVD, multimedia applications for language education have become increasingly common. These applications often contain extensive textual material in

addition to video, which allow language lessons to be contextualized in a way not possible with video alone. Often, this supplemental content serves to exemplify not only the language, but associated cultural knowledge as well (Kramsch and Andersen 1999). Such contextualization enables the learner to better function in the target culture, as they acquire not just the linguistic skills, but the sociolinguistic knowledge necessary to use the new language effectively.

A disadvantage of this type of rich content is its accompanying cost: it is often difficult and time-consuming to create. First, the video content itself must be acquired through some means, either licensed from a third party or custom-filmed to capture a particular pedagogical point; both options have significant associated costs (A. K. Melby, personal communication, June 19, 2008). Both also have inherent limitations: third-party video usually offers little in the way of contextual material, may require substantial work on the part of the content developer, and may be difficult or impossible to license. Conversely, custom-filmed video is difficult to produce in such a way that it captures the viewer's interest and attention in the same way as content produced solely for its entertainment value. The Electronic Film Review (EFR) approach takes some of the best of both methods: feature films are used, which keeps development costs lower and helps to engage the learner more fully in the process (EFR Team, Brigham Young University 2006). Further, because the original DVD content is untouched, there is no need to acquire a license in order to use it. Such an approach provides the content developer two significant advantages: costly licensing fees are avoided for those films where a license is possible, and the library of films from which to choose is broadened to include those for which licensing is not available. (Bush, et al. 2004).

Even so, the creation of contextual material is nonetheless a costly and time-consuming proposition, with a typical film requiring close to 200 man-hours of work by a content developer specially trained in the annotation process (Keeler 2005). In order to be of the most use to the widest possible audience, the EFR system needs an authoring suite with a gentler learning curve, one in which the process of annotation

is facilitated through automated means wherever possible, and made more intuitive when automation is impractical.

1.3. CHALLENGES TO THE EFR APPROACH. There is still some question about the best way to provide contextual, annotative content to the learner. First, it should be noted that there are differences of opinion regarding the best way to present the annotative materials: concurrent with the video segments, or prior to their playback (A. K. Melby, personal communication, August 27, 2009). Secondly, the types and volume of content must be determined, often informed by the overall educational intent of the package: vocabulary building, acculturation, grammar instruction, and listening comprehension are some common choices. Additionally, there is an open discussion at present about whether specialized, context-sensitive glossaries provide any benefit over a simple learner-directed dictionary lookup, that is, whether the curriculum builder/software should indicate the correct word sense used in a given segment, or whether it should be left to the language learner to determine from context.

Another challenge should be mentioned as well, and it bears discussion. When teaching a language, either to a classroom full of students or in a one-on-one scenario, two markedly different approaches are possible. The first methodology, exemplified by English as a Second Language (ESL) programs, assumes no knowledge of the learner's primary language: all instruction, interactions, and supplementary materials are in the target language. This is the approach taken by Keeler in her 2005 effort, and is perhaps the more difficult of the two in terms of content creation. Indeed, the generation of broadly comprehensible annotations, restricted to the language of instruction, is beyond the capability of current automated systems. The second instructional approach differs primarily in one key aspect: the primary language of the learner(s) is taken into account, and can be leveraged in the annotative materials. Programs in English as a Foreign Language (EFL), presently common across much of Asia, are an application of this approach (Cautrell 2007). An evaluation of the relative merits of these two approaches is beyond the scope of this project, but it suffices to say that both are presently regarded as legitimate endeavors.

This project focuses on annotation for the purpose of vocabulary enrichment, with context-appropriate glossary entries presented concomitantly with the video content, with annotative materials in both the learner's primary and target languages. This narrowed emphasis will allow for a greater level of sophistication in the resulting tool, which in turn should lead to greater improvements in the efficiency of the overall authoring process. An evaluation of the efficacy of this particular pedagogical focus over the others previously mentioned is outside the scope of this project, and is left to future researchers.

As has already been stated, the creation of engaging content to accompany feature films is as yet a laborious process, one that demands more time than most instructors are willing or able to spend (Dennis Packard, personal communication, September 2007). The overall annotation process can be broken down into three main tasks:

1) partitioning of the film into linguistically and/or narratively significant segments,
2) identification of vocabulary items within these segments, and
3) development of explanatory content for each of these vocabulary items.

These tasks each bear challenges in the current generation of authoring tools; film partitioning, for instance, requires the hand-creation of an EFR segmentation script, commonly referred to as a SEGFILE. Each segment must be transcribed, and any LEXICAL ITEMS of interest (e.g. terms, idiomatic expressions, etc.) must be flagged for vocabulary treatment. Explanatory material must then be developed and linked to each instance of the vocabulary item, which, as lexical items may comprise more than single words, will involve more work than a simple dictionary lookup. The current software used to accomplish these tasks can best be described as byzantine, and the associated learning curve for content developers is quite steep. Further, the annotation process itself is tedious, prone to error, and requires the use of an overly restrictive scripting language. Where visual tools exist, they are unconnected to the content compilation tools, which depend on the well-formedness of hand-coded data files. Finally, reuse from one film to the

next is not facilitated, requiring a content developer to start over from scratch with each successive film so treated.

In summary, the overall EFR content creation process—though functional in its current state—has much room for improvement.

1.4. PROPOSAL. The difficulties outlined in the previous section can all be lessened through the judicious use of several current resources and technologies: segmentation can be approximated by leveraging existing online resources, vocabulary identification and EFL-style treatment can be accomplished via terminology lookup and statistical machine translation, and reuse achieved through use of a centralized terminology management system. Together, these new tools can form the basis for a complete authoring suite, one that will facilitate the overall process of metadata creation, reduce development time, and smooth the learning curve required of the end user.

At a high level, the project flow will be as follows, given a film that the content developer wishes to treat:

1. Produce EFR-segmentation file for the film (generated from publically-available resources where possible)
2. Content developer edits this segfile to mark up all instances of lexical items to receive contextual glossary entries
3. Produce a terminology data file for system use, either through database export or via automated transformation of user-edited spreadsheet
4. Parse the segfile, identifying all marked-up lexical items of interest
5. Query the terminological data store for the treatment of these lexical items
    a. If no match is found, queue the item for processing by the machine translation engine

6. Run all queued sentences through the machine translation engine, capturing the context and suggested translation for the lexical items of interest
7. Merge the output of the terminology lookup and machine translation steps
8. Use this combined output file to produce context-sensitive glossary web pages, using the existing EFR Vocabulary Tool.
9. Use the output of the machine translation process to enrich the terminological data store

A more detailed discussion of this implementation will be described in the following chapters.

1.5. CRITERIA FOR MEASURING SUCCESS. We will use four main criteria to measure the success of the project:

1) provide a full replacement for all current functionality,
2) automate the lookup of lexical items in a termbase to see if they are already covered
3) automate the selection of context-sensitive translations for those items not already treated
4) provide a mechanism for enriching the termbase with the new suggested translations

If these criteria are met, then we will deem the project a success. Moreover, it is anticipated that this change in methodology will lead to greater efficiencies over time, as the growth of the shared vocabulary database will lead to faster term extraction cycles, and reuse of existing database entries will result in much faster vocabulary item treatment, as this will be as quick as creating a hyperlink or performing a database lookup. Because this growth will likely take some time before reaching a point where it has such an effect, an evaluation of its efficacy will be left to future researchers.

1.6. ALTERNATIVE APPROACHES. It should be noted that the tools we use to accomplish these tasks are not the only possible solutions. For instance, film segmentation could be done via visual pattern recognition and scene detection (Lin and Zhang 2000). Film transcription and transcript time-alignment could be accomplished through Optical Character Recognition (OCR) of the film's subtitles (Li,

Doermann and Kia 2000) or direct extraction of closed-captioning (line 21) data (Zhu, Toklu and Liou 2001). Both of these were considered, then ultimately rejected due to the inherent inaccuracy of many subtitle and closed-captioning streams, which often do not closely match, but only approximate, the audio content.

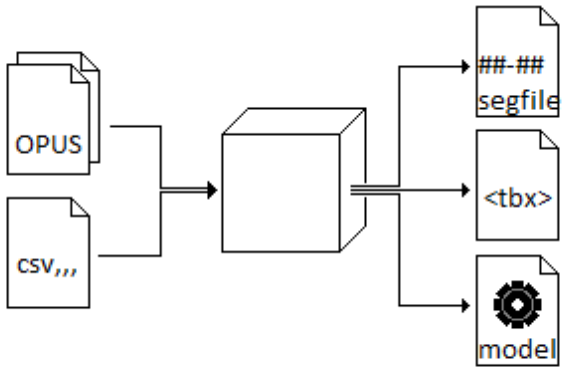
Finally, in recent years researchers have demonstrated a technique termed Shadow Speaking, in which an observer watching a film repeats the dialog into a sound-canceling microphone, which is then processed by an automatic speech recognizer (Boulianne, et al. 2006). This approach, although proven to be quite effective, was deemed unsatisfactory for an end-user application such as the one intended for this project: it seems unlikely that a content developer would want to shadow-speak an entire film prior to being able to annotate it, and even the more technically inclined would likely prefer to focus on the linguistically-focused aspects of film treatment.

1.7. SUMMARY. As the nations of the world become more and more closely interconnected, the need for multilingual education grows increasingly urgent. In order to be most effective, the need for this educational content to engage and retain the student's attention grows as well. In turn, as the need for engaging content expands, more efficient production tools will be required to keep up with the demand. It is in this spirit that the present effort has been undertaken, with the hope that its fruits will be used to better prepare the rising generation to meet the demands of a global society.

The remainder of this paper will be as follows: Chapter 2 will discuss the data preparation steps undertaken to ensure clean and usable data inputs, and provide an overview of the process of creating language/phrase models for the machine translation engine. Chapter 3 will describe the twin processes of terminology lookup and machine translation of lexical items. Chapter 4 will outline the process for iteratively improving the terminological data store by incorporating the results of the translation step on each subsequent invocation. Finally, Chapter 5 will discuss the results of the project, and its implications for future work in the field.

CHAPTER 2: DATA PREPARATION

As with any project, the quality of the program's outputs is limited by the quality of the inputs; the data preparation phase of the project concerns the creation of good inputs. This involves several subtasks: film transcript alignment/segmentation, lexical item identification, seeding of a preliminary terminology database, and training the statistical machine translation system.



Step 1: Data Preparation

2.1. FILM SEGMENTATION. The problem this poses is twofold: first, time codes that mark the beginning and end of each video segment must be defined, and second, the linguistic content of these segments must be transcribed. There are multiple approaches to fully automating this task, many of which were mentioned in section 1.6. For the purposes of this project, however, we have chosen to leverage freely-available online resources to provide the basis for film segmentation. The OPUS open-source parallel corpus, created by Jörg Tiedemann, includes, as one of its main resources, the OpenSubs corpus (Tiedemann 2009). Originating from the popular Internet web site OpenSubtitles.org, this corpus comprises 361 bitexts in 30 languages, and is freely available for download. Included are a collection of BITEXTS, parallel text files in which sentences in the source and target language are aligned with one another. These are encoded in the XCES (XML Corpus Encoding Standard) format, and represent time-

aligned, segmented transcript information for all the films included in the OpenSubs corpus. Figure 1 shows a sample segment from one of these bitexts:

Figure 1: Sample XCES Segment

```
<s id="331">
  <time id="T228S" value="00:16:29,019" />
  <w id="331.1">The</w>
  <w id="331.2">most</w>
  <w id="331.3">famous</w>
  <w id="331.4">pirate</w>
  <w id="331.5">in</w>
  <w id="331.6">his</w>
  <w id="331.7">time</w>
  <w id="331.8">.</w>
</s>
<s id="332">
  <w id="332.1">My</w>
  <w id="332.2">dad</w>
  <w id="332.3">told</w>
  <w id="332.4">me</w>
  <w id="332.5">all</w>
  <w id="332.6">about</w>
  <w id="332.7">him</w>
  <w id="332.8">.</w>
  <time id="T228E" value="00:16:33,092" />
</s>
```

At first glance, the XCES appears to only contain two useful pieces of information: `<s>entences`, which in turn contain `<w>ords`. The temptation, then, is to treat the `<s>` elements as segments as well. However, upon closer inspection, we see that the first sentence begins with a `<time>` element, and the second sentence ends with one. Additionally, the two time elements have matching *id* attributes–T228S and T228E, respectively–which seem to refer to the Start and End timecodes of a segment named T228). Experimentation with these timecodes in the legacy EFR segmentation tool reveals that these timecodes do, indeed, define the boundaries of the dialog in question. Armed with this knowledge, we can now convert XCES segments like Figure 1 into EFR segfile blocks as shown in Figure 2:

Figure 2: Film Segment in EFR Segfile Format

```
29641-29763 (00:16:29:01 - 00:16:33:03)
Scene 228
```

```
//T: The most famous pirate in his time. My dad told me all about him.
```

With this understanding of the precise nature of the data structures encoded in these XCES files, a better conversion into EFR-segfile is possible. Initially, since the starting point was an XML-based format, an attempt was made to use XSLT transformations to accomplish the reformatting. However, XSLT has very poor support for string manipulation, which is important due to the not-insignificant differences between the XCES time format and the EFR-segfile format. See (1) and (2) for a comparison:

(1) XCES timecode: 00:16:29,019

(2) EFR-segfile timecode (frame count): 29641

EFR-segfile timecode (SMPTE timecode): 00:16:29:01

Timecodes in XCES are represented by hh:mm:ss,mms (hours, minutes, seconds, and milliseconds). EFR-segfile uses two formats: the first is a raw frame count, useful for low-level DVD access; the second is the similar hh:mm:ss:ff (hours, minutes, seconds, frames) format commonly used in media players. At the standard conversion rate of 30 frames per second, both numbers are straightforward to derive in any language capable of evaluating math expressions:

(3) 019 ms = 0.019 sec * 30 frames/sec = 0.57 frames ≈ 01 frames

00:16:29,019 → (29.019 sec) + (16 min * 60 sec/min) = 989.019 sec

989.019 sec * 30 frames/sec ≈ framecode 29671

For all of its utility, XSLT isn't capable of that kind of mathematical feat. Therefore, the decision was made to use Python, a robust, mature scripting language available on most modern platforms. The resulting script, `xces2seg.py`, takes an XCES file as input and produces a correctly-formatted EFR-segfile as output. This script, along with all other project scripts, is included in Appendix I: Source Code.

One final note on segmentation: for films not explicitly included in the OpenSubs corpus, the OPUS project provides a set of command-line Perl scripts to convert raw subtitle files—such as those available on OpenSubtitles.org—into new XCES-formatted data files. Once in XCES format, these files can then in turn be transformed into EFR-segfile format with the `xces2seg` tool.

2.2. FLAGGING LEXICAL ITEMS. Once a preliminary segfile has been generated for a film, the content developer will need to manually review each segment to identify the lexical items of pedagogical interest. For the initial implementation of the system, this is still a manual process: the segfile must be opened in a text editor, and the desired lexical items marked up with pseudo-XML start and end tags, as in Figure 3:

Figure 3: EFR Segfile Segment with Markup

```
29641-29763 (00:16:29:01 - 00:16:33:03)
Scene 228
//T: The most famous <item> pirate </item> in his time. My dad told me all about
him.
```

These tags will be used by the segfile processor to generate the glossary and vocabulary helps for the student. This process will be discussed in more detail in the next chapter.

2.3. TERMBASE CONSTRUCTION. More often than not, a content developer just beginning to treat films for instructional use will not have a database of those lexical items they want to present to their students as part of the film screening. Such a database of TERMS[1], hereafter referred to as a TERMBASE, provides a centralized location for maintenance and access purposes, which can greatly speed up the process of glossary creation. Once a term has been identified, documented, and stored in their termbase, a content developer can simply refer back to that entry every time they want to add contextual help to an instance of that term in the transcript. Further, lexical items referencing a given term or concept need not be restricted to single words, nor must term instances only take a single lexical form.

[1] For the purposes of this project, we define 'term' as a word or group of words carrying a specific semantic concept.

Many termbase applications have a proprietary data representation, but users often need to share terminological data across group, company, and even international lines. The ISO Term Base eXchange (TBX) standard was developed for just this reason (LISA 2008). As an XML application, TBX is verbose, but versatile: instead of a single schema, it actually defines a family of Terminological Markup Languages, or TMLs, which cover just about every conceivable type of terminological information, and which can be added to when a particular need cannot be met by any existing TMLAdditionally, since it is an XML-based format, TBX can be automatically queried, merged, and transformed into a variety of termbase import formats. This makes it an ideal tool for use in our system: a content developer with a termbase can export all or a subset of their terminology into a TBX representation, and provide it to the application for lookup of a transcript's lexical items of interest.

However, the verbosity of the TBX family of formats can be daunting to a curriculum developer that does not yet have a terminology management solution. In order to take advantage of the benefits of terminology lookup, these individuals would have to hand-create and maintain their data directly in raw TBX. To help address this need, the Localization Industry Standards Association (LISA) has developed a simplified form called TBX-Basic (A. K. Melby 2009), which includes the most frequently used data categories for translation purposes. This simplified format also includes a non-XML representation dubbed MRC[2] TermTable, which can be transformed losslessly into the XML form. Nevertheless, even this format may be overly complicated for some users.

The original EFR tool chain that this system is replacing utilizes a much simpler glossary format: a Microsoft Excel spreadsheet, one row per concept, one lexical item each for source and target language, and a few additional data categories that apply to the concept as a whole. Table 1 contains an extract of one such spreadsheet, used in the ESL treatment of the film Holes (note that some columns have been omitted for brevity's sake):

[2] Multiple Rows per Concept; see ttt.org/tbx for more information.

Table 1: Simple concept-oriented vocabulary spreadsheet

Concept ID	Head Word	Grammar	Definition
best_served.v.1	best served	verb	To be "best served" means that something "will help the most."

Using this original table layout as a basis, and given the need to quickly add a large number of rows to the termbase, a very simple flattened data layout was devised that could be transformed into MRC format, and from there into TBX-Basic. Table 2 shows an example of this flattened format, Figure 4 shows its corresponding MRC TermTable representation, and Figure 5 shows a fragment of TBX corresponding to those same rows.

Table 2: Flattened Terminological Data

subjectField	partOfSpeech	en-us	fr-fr
General	Adjective	fabulous	superbe
General	Noun	feet	pieds

Figure 4: Flattened Terminological Data in MRC Format

```
C013   subjectField  General
C013en1       term   fabulous
C013en1       partOfSpeech  adjective
C013fr1       term   superbe
C013fr1       partOfSpeech  adjective

C014   subjectField  General
C014en1       term   feet
C014en1       partOfSpeech  noun
C014fr1       term   pieds
C014fr1       partOfSpeech  noun
```

Figure 5: Flattened Terminological Data in TBX-Basic Format

```
<termEntry id="C013">
  <descripGrp><descrip type="subjectField">General</descrip></descripGrp>
  <langSet xml:lang="en">
    <tig id="C013en1">
      <term>fabulous</term>
      <termNote type="partOfSpeech">adjective</termNote>
    </tig>
  </langSet>
  <langSet xml:lang="fr">
    <tig id="C013fr1">
      <term>superbe</term>
      <termNote type="partOfSpeech">adjective</termNote>
    </tig>
  </langSet>
</termEntry>
<termEntry id="C014">
  <descripGrp><descrip type="subjectField">General</descrip></descripGrp>
  <langSet xml:lang="en">
    <tig id="C014en1">
      <term>feet</term>
      <termNote type="partOfSpeech">noun</termNote>
    </tig>
  </langSet>
  <langSet xml:lang="fr">
    <tig id="C014fr1">
      <term>pieds</term>
      <termNote type="partOfSpeech">noun</termNote>
    </tig>
  </langSet>
</termEntry>
```

The conversion from this flat format to MRC TermTable was accomplished via a simple Python script, `flat2mrc.py`. This file was then transformed into TBX-Basic via the LISA TBX tools.

Another note about this data: the items included all have a subject field of *general*, as they are extracted from film transcripts that aren't typically constrained to a narrow domain. This potentially makes the extracted terminology less useful than if it were domain-specific; in practice, the more precise the domain, the more accurate the produced glossary entries will be.

2.4. TRAINING MOSES. The final data preparation task to be done is the creation of statistical models for the machine translation engine. The system targeted by the project is Moses, an open-source, statistical machine translation application frequently used within the academic community. Two distinct toolkits

exist for creating Moses-compatible models: IRSTLM, developed by the Fondazione Bruno Kessler, and SRILM, produced by SRI International. These tools, as well as Moses itself, were intended to be used primarily in a UNIX/Linux environment, but can be made to run in both Apple's OS X operating system and Microsoft Windows (via the Cygwin emulator). The Moses online documentation covers the setup process in Linux and OS X, but fails to detail the steps necessary to get everything running correctly in Windows/Cygwin. See Appendix II for a detailed list of the changes that need to be made to accomplish this.

Once the tools have been built, they can be used to generate a LANGUAGE MODEL and PHRASE MODEL, which may be a bit misleading, as they do not represent actual linguistic data, but rather, statistical information about the language and phrase constructions present in the corpus of data used to train the system. These models can be tuned to meet particular needs. For example, if a content developer wishes to treat a series of films within a narrow linguistic domain (e.g. movies about baseball), a training corpus extracted exclusively from other such films (Field of Dreams, The Rookie, etc) could be used to provide more accurate results from the decoder.

The first step is to prepare the data to be used for training inputs. Here again we will be leveraging work produced by Tiedemann as part of the OPUS project: each of the language pairs in the OpenSubs corpus can be downloaded in a format easily processed by the language/phrase modeling tools. This process will be clearest with a detailed chronology of the commands run from within a Cygwin Bash shell, so the following narrative will be interspersed with blocks representing these command sequences.

Begin by creating a folder hierarchy within the main Moses directory, if these folders do not already exist:

```
$ mkdir work/opus
```

Next, download the `eng-fre.txt.gz` sentence alignments from the OPUS/OpenSubs website (located at http://urd.let.rug.nl/tiedeman/OPUS/OpenSubtitles.php at the time of this writing).

Uncompress `eng-fre.txt` into work/opus, and split it into parallel English and French text files, per Moses' training data requirements. OPUS provides a Perl script for this purpose, but we've provided a Python script as well[3]:

```
$ python linguassist/opus2moses.py work/opus/eng-fre.txt work/opus/corpus/eng-fre en
fr
```

After matching up the English and French sentences and discarding any that don't sync up, 216,418 sentence pairs remain in the eng-fre.en and eng-fre.fr files:

```
$ wc -l work/opus/eng-fre.en work/opus/eng-fre.fr
  216418 work/opus/eng-fre.en
  216418 work/opus/eng-fre.fr
  432836 total
```

The next step is to tokenize and normalize the input files, using the Moses scripts:

```
$ mkdir work/opus/corpus && mkdir work/opus/lm
$ cat work/opus/eng-fre.en | tools/scripts/tokenizer.perl -l en >
work/opus/corpus/eng-fre.tok.en
$ cat work/opus/eng-fre.fr | tools/scripts/tokenizer.perl -l fr >
work/opus/corpus/eng-fre.tok.fr
$ tools/scripts/lowercase.perl < work/opus/corpus/eng-fre.tok.en > work/opus/lm/eng-
fre.lowercased.en
$ tools/scripts/lowercase.perl < work/opus/corpus/eng-fre.tok.fr > work/opus/lm/eng-
fre.lowercased.fr
```

The Moses training walkthrough recommends filtering out all long sentences via the `clean-corpus-n.perl` script. However, given the brief nature of most sentences in the OPUS corpus, this is unnecessary.

Once the data has been properly prepared, we can proceed to build the language model, using SRILM:

```
$ tools/srilm/bin/cygwin/ngram-count -order 3 -interpolate -kndiscount -unk -text
work/opus/lm/eng-fre.lowercased.fr -lm work/opus/lm/opus.lm
```

This process results in a modest number of n-grams, but enough to give the system a decent workout:

```
$ head -n 5 work/opus/lm/opus.lm

\data\
ngram 1=49398
ngram 2=356029
```

[3] As with the rest of the project's source code, opus2moses.py is included in the Appendix.

```
ngram 3=168242
```

Now, we train the phrase model. As this process can take quite a long time and put a strain on the CPU, the training walkthrough suggests running it in the background and via the `nice` priority scheduler. (Note also that the training walkthrough identifies this script as `train-factored-phrase-model.perl`, but in recent exports from the central repository, it seems to have been renamed to simply `train-model.perl`):

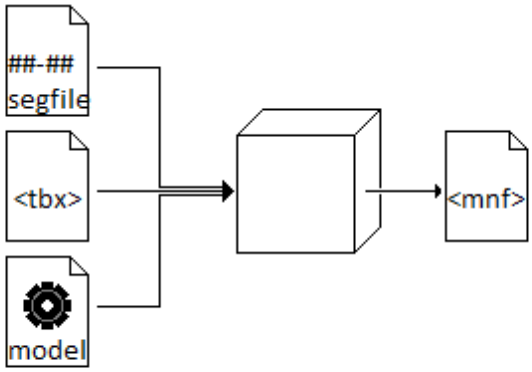
```
$ nohup nice tools/moses-scripts/scripts-20100605-2158/training/train-model.perl -
scripts-root-dir tools/moses-scripts/scripts-20100605-2158/ -root-dir work/opus -
corpus work/opus/corpus/eng-fre.lowercased -f en -e fr -alignment grow-diag-final-and
-reordering msd-bidirectional-fe -lm 0:3:/cygdrive/c/LingMA/moses/work/opus/lm/opus.lm
>& work/opus/training.out
```

The end result, after about an hour of processing, will be complete language and phrase models in the work/opus folder. We will return to these models in the next chapter.

CHAPTER 3: TERM LOOKUP AND TRANSLATION

Once the input formats and Moses training data are ready, the second task to be addressed by the new system is the identification and treatment of vocabulary items of instructional interest.

At a high level, this module has three inputs and a single output format:



Step 2: Term Lookup and Translation

We can visualize Step 2 as a black box taking in an EFR segmentation file, a TBX file of terminological data, and a Moses configuration file pointing to the language and phrase models generated in Step 1. The system analyzes these inputs and produces one or more output files in a specialized TBX TML developed specifically for this project, MNF (Mapped iNterchange Format). This format is basically a superset of TBX-Basic, with a few custom data categories to maintain information linking term concepts with their in-context lexical items. This will make the process of contextual glossary generation in Step 3 much simpler, as the outputs of both the term-lookup and machine-translation processes will retain information about which segments contain which lexical items. Inside this black box, this module actually consists of three distinct processes:

1. Term lookup
2. Machine translation of the lexical items of interest not found in the termbase

3. Post-processing of the machine translation output

3.1. TERM LOOKUP. Once the data sources are prepared, the first step to producing a new glossary is to process the EFR segfile, looking for those lexical items identified by the instructor to receive glossary treatment. This is done via the Linguassist.py script, the full source of which is located in Appendix 1.

Linguassist performs three basic steps:

1) Parse the EFR Segfile and identify all marked-up lexical items of interest
2) Load the TBX file and query it for matches from these items
3) Output the list of terms it found, and the list of items it failed to find

The Segfile parsing is done by a helper module, `segfile/parser.py`. Its algorithm is pretty straightforward: for each film segment it finds, it looks for the pseudo-XML `<item>` start and end tags. If found, it extracts the lexical item embodied by these tags and indexes it for easy retrieval later on. It also stores the full transcript for that segment, as well as its timecode and title, in case these are needed later.

The TBX parsing is also accomplished by a helper module, `TBX/parser.py`. As with the Segfile parser, it is quite straightforward. In fact, the current incarnation is perhaps overly simplistic in its TBX parsing, and will fail to capture any additional information beyond source and target terms, languages, and subject field. Future iterations of the parser should capture the full breadth of information contained in the provided TBX file, whatever its source. That said, this version captures the essential information.

Once Segfile and TBX are parsed, the rest is trivial: Linguassist iterates through the list of items of interest provided by the Segfile, and queries the TBX class for a match. This querying mechanism isn't sophisticated, but simply does a lexical string match; future versions would do well to implement some level of BASE FORM REDUCTION (BFR)[4]. If a match is found, the term and its originating context are

[4] Without BFR, the lookup is constrained to the specific lexical representation found in the source text. More accurate results can be achieved by including references to a term's alternate forms in the termbase.

output to a 'hit-list' file in MNF format, which can be merged with the MNF file produced by the Moses post-processing; see section 3.3 for more on this.

Any lexical items which cannot be found in the provided TBX termbase are output to a 'missed-list' flat file, in a format suitable for automated translation via the Moses decoder.

3.2. MACHINE TRANSLATION. Moses operates on files in the same basic format as the language modeling tools discussed in Chapter 2: Data Preparation: plain text, one sentence per line, tokenized and normalized to lower case. The Moses input file generated by Linguassist has two additional features that will be important when post-processing the Moses output: first, the term of interest is identified by XML-style start and end `<item>` tags, so it can be extracted from the output translation (we keep the lexical item in its full sentence context to give Moses enough information to translate more accurately). The second addition to the standard input format comes at the end of each line: `<src>`, another XML tag. This tag contains a comma-delimited list of segment IDs where that lexical item is found in the original text. This is how the software determines which glossary entries to highlight during any given segment of the film, as we will see in Chapter 4: Iteration and Improvement.

Figure 6 shows a sample line from a Moses input file that contains this additional XML markup, along with its corresponding output line:

Figure 6: Sample Moses input and output line with XML markup

```
The most famous <item> pirate </item> in his time . My dad told me all about him .
<src>225,228,1261,1263,1448,1455</src>
```

```
le plus célèbre <item> pirate </item> dans son temps . mon père m ' a raconté son
histoire <src>225,228,1261,1263,1448,1455</src> .
```

Note that the two tags make different use of whitespace: the start and end `<item>` tags don't touch the lexical item they contain, or the text on either side thereof. The `<src>` tag is offset from the rest of the

sentence, but its content must *not* contain any whitespace, lest it be treated as text to be translated. This is due to the way Moses handles XML markup. The `-xml-input` command-line parameter can explicitly direct the decoder to include, exclude, or ignore any markup, but by default, XML tags are treated as `pass-through`. That is, Moses will ignore the tags when translating, but attempts to re-insert the markup in the appropriate places in the output sentences. That means that the translation of the lexical item should still be marked with `<item>` tags, making it easy to parse out of the full sentence, and the <src> tag should still be at the end of the string, leaving the sentence uninterrupted. In the next section, we will make use of both of these features.

3.3. POST-PROCESSING OF MOSES OUTPUT. Once the Moses decoder has given its best efforts to translate the input sentences it was given, we feed the input and output files into our next tool, Moses2MNF. This script lines up the source and target sentence pairs, finds the marked-up term in each, and uses them as the basis for a collection of new `<termEntry>` elements, as exemplified in Figure 7:

Figure 7: Moses input, output, and the resulting MNF

```
// Moses input sentence:

setting <item> booby traps </item> so anybody who tried to get in would die .
<src>254,255,787,788,792,809,971,1338,1340,1341</src>
```

```
// Moses output sentence:

<item> des pièges </item> si quelqu ' un qui a essayé de mourrait .
<src>254,255,787,788,792,809,971,1338,1340,1341</src>
```

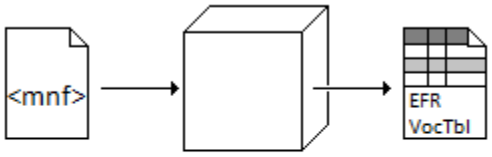
```
// Generated MNF term entry:

<termEntry id="booby_traps">
   <descrip type="subjectField">General</descrip>
   <admin type="sourceSegment">254</admin>
   <admin type="sourceSegment">255</admin>
   <admin type="sourceSegment">787</admin>
   <admin type="sourceSegment">788</admin>
   <admin type="sourceSegment">792</admin>
   <admin type="sourceSegment">809</admin>
   <admin type="sourceSegment">971</admin>
   <admin type="sourceSegment">1338</admin>
   <admin type="sourceSegment">1340</admin>
   <admin type="sourceSegment">1341</admin>
   <langSet xml:lang="en">
      <tig>
         <term>booby traps</term>
```

```
         <descrip type="context">setting booby traps so anybody who tried to
get in would die .</descrip>
      </tig>
   </langSet>
   <langSet xml:lang="fr">
      <tig>
         <term>des pièges</term>
         <descrip type="context">des pièges si quelqu ' un qui a essayé de
mourrait .</descrip>
      </tig>
   </langSet>
</termEntry>
```

These MNF files contain all of the information necessary to create simple contextual glossaries for an EFR: term representation in the source and target languages, an in-context usage of that term in each language, and references to the segment IDs where those lexical items originated. This process will be discussed in greater detail in the next chapter.

CHAPTER 4: ITERATION AND IMPROVEMENT

Once the main tasks of termbase lookup and machine translation are finished, all that remains is to make use of the results.



Step 3: Result Processing

As discussed in Chapter 3: Term Lookup and Translation, the main outputs of the processes comprising Step 2 are one or more MNF files, representing the translations of the lexical items identified by the instructor for glossary treatment. To make use of these entries, they must first be converted into a format usable by the rest of the EFR authoring toolchain. Additionally, we want to load the results from the Moses translation piece back into the centralized TBX data store in order to enable quicker and more accurate glossary creation in subsequent iterations.

4.1. CONVERTING TO VOCTABLE FORMAT. With the MNF data generated in Step 2, the process of building the EFR VocTable is quite straightforward. As discussed in Chapter 2: Data Preparation, the EFR VocTable format is row-based, each row indicating a unique entry. The basic format includes segmentation information for the film being treated, one word on each line. If that word constitutes the HEAD WORD of a lexical item, the row then includes an additional 9 columns of terminological information about that item (or simply a previously-defined term's concept ID). When MNF data is converted to this format, it should look something like Table 3:

Table 3: Abbreviated VocTable CSV

Chapter#	Token#	Token	Utterance ID	Concept ID		Translation		Images
1	1	pirate	T225	pirate.1		pirate		pirate.png
1	2	pirate	T228	pirate.1	…		…	
1	3	pirate	T1261	pirate.1				
		…				…		…
2	7	booby	T254	booby_traps.1		des pièges		trap.png
2	8	booby	T255	booby_traps.1	…		…	
2	9	booby	T787	booby_traps.1				

The original VocTables created by the EFR system had complete coverage for the film transcript, covering every token in the entire film from beginning to end. However, this was more as a help to the content developer than it was necessary to the glossary-generation code. The tables produced by MNF2VocTable, consequently, include only those rows that directly contribute to the glossary output. Even with this difference, however, the resulting VocTable CSV file is fully compatible with the legacy EFR system's glossary generator. Future efforts may replace that application as well, but such efforts are outside the scope of this project.

4.2. RE-SEEDING THE TBX DATA STORE. If a commercial software package is being used for terminology management, this is quite straightforward. Since the MNF format is a superset of TBX, it can be easily re-ingested by the termbase, integrating any of the outputs from Moses that the content developer feels are of sufficient quality. A new export from the termbase can then be made for any subsequent EFR film treatment, which will enable the content developer to capitalize on the results of the previous generation.

It is expected that this process of continually re-seeding the termbase with machine-produced, human-approved result sets will, over time, have a significant impact on EFR development time.

CHAPTER 5: CONCLUSIONS

5.1. ANALYSIS OF RESULTS. This project was envisioned as a way to reduce the time and effort involved in the creation of EFR film annotations. The logical question to ask at this point is, did it accomplish that goal?

The first aim of the project was to "provide a full replacement for all current functionality." This has been accomplished, as anything that could be done manually previously (film segmentation, term lookup, translation suggestion) has a counterpart in the new system.

The next goal of the project was to automate the lookup of lexical items in a termbase. As Section 3.1 described, this has been achieved in a limited form, albeit without any kind of base form reduction, which would render the lookup much more intelligent. A possible workaround was suggested, in which all inflected forms of a lexical items are included as alternate forms for the concept entry, but this is an imperfect solution at best.

Third, the project set out to automate the suggestion of context-sensitive translations for lexical items in need of treatment. This has been accomplished via the Moses invocation methodology outlined in Sections 3.2 and 3.3.

The final goal of the project was to create a means whereby a termbase could be enriched via incorporation of machine-suggested translations. As Section 4.2 indicated, this is accomplished via the tool chain's ability to convert the Moses input/output files into a TBX-compatible format, which can then be easily imported into most terminology management packages. This is perhaps the most powerful and immediately useful of the tools created as part of this effort.

5.2. PROGRAM PORTABILITY. The scripts created for this project were written in such a way that many of them can be run independently, without needing to commit to the entire tool chain as a whole. Some examples may serve to best explain under what conditions this would be fruitful.

The `XCES2Seg` tool, written to quickly convert an OPUS transcript file into EFR-segfile format, is useful for anyone looking to create an Electronic Film Review for a film covered by the OpenSubs corpus, whether or not they intend to use it for targeted language instruction. As a standard, EFR was created for many types of customized video playback. Beyond the "study lab" application of immediate concern to this project, the software has been successfully used in classroom lectures, theater screenings, and even private home viewings. The ability to quickly create a starting EFR is therefore of interest to a broader audience.

Similarly, the `CSV2MRC` tool enables the efficient creation of a barebones TBX-Basic file, which could be used to quickly seed a beginning termbase, or any other purpose for which a TBX data file could be used. This therefore provides a useful entry point for newcomers to terminology management, who may have yet to commit to a full-fledged termbase solution.

In the course of tool creation, standalone parsers for both TBX and EFR-Segfile were produced. These could both be reused in future Python applications or command-line-based toolchains where either of those formats is required, which little to no modification necessary.

Finally, the `XLIFF.py` module, which contains conversion routines for both OPUS-to-XLIFF and XLIFF-to-Moses transformation paths, could be reused when interaction with XLIFF (XML Localisation Interchange File Format) data is desired. A standard produced by the Organization for the Advancement of Structured Information Standards (OASIS), XLIFF has significant support within the localization community, and this module allows developers to move data in and out of that format. Localization data originally in an unrelated format could, for example, be used to generate Moses training data that may be a better fit for a given application than the data in the OpenSubs corpus. The `XLIFF.py` module could also be used to transform OpenSubs data files into XLIFF for use in an application wholly unrelated to EFR creation.

5.3. FUTURE WORK. Although, as has been mentioned already, film transcripts are often quite difficult to treat, part of the data preparation and term identification process could be automated with some simple segment processing. For instance, a segfile pre-processor could be created to automatically recognize and flag certain classes of items such as proper names, ordinal numbers, expletives and other STOP WORDS (i.e. those words in a language whose purpose is largely pragmatic, rather than informational). This would potentially make it easier for a future system to offer suggestions about lexical items to be treated, rather than relying solely upon the instructor to identify these manually.

Another simple improvement to the segfile processor would be to have it scan the text for repeat uses of lexical items identified by the instructor in previous segments. Thus a given lexical item would only need to be flagged once in order to be presented to the student at every subsequent occurrence throughout the film. The benefit of such a pre-processor would, however, hinge on its ability to correctly determine different contextual meanings for the same lexical item.

The benefits of the terminology lookup step could be increased dramatically if coupled with a good base form reduction algorithm. This would eliminate the need for inclusion of all alternate forms in the termbase's concept entries, which in turn would result in less redundancy in any produced glossary entries.

A similar performance improvement could be achieved if, instead of relying on user-supplied TBX files, the system could access a centralized, internet-accessible termbase (Melby, Fields and Carmen 2006). Once vocabulary items have been identified—by either the system or the user—a database lookup could be performed. If that lexical item has been previously treated and uploaded by anyone, the software can alert the content developer to its availability. They can then simply link to the existing treatment, saving the user the trouble of writing it themselves, or they can create an alternate treatment if desired. The instructor may elect to upload the information they create back into the central termbase, thus facilitating the treatment of future instances of that same vocabulary item by others. Over time, as the

central termbase grows, one could reasonably expect that the term lookup and definition steps will become more and more efficient, as each new user will be able to leverage the work of those that have come before.

Finally, the benefits of the new system outlined herein should be rigorously evaluated by those qualified to do so: content developers. Specifically, two points of evaluation could and should be made by future researchers:

1) Is this new system more user-friendly, that is, can curriculum developers be trained more quickly and effectively in its use?
2) Does this new system lower the barrier to entry for EFR creation, allowing more content developers to more easily produce their own Electronic Film Reviews?

Ultimately, if the EFR approach is to be successful in the long term, these questions must be addressed. It is the author's hope that this project will be a positive step in that direction.

REFERENCES

Boulianne, G., et al. "Computer-assisted closed-captioning of live TV broadcasts in French." *Ninth International Conference on Spoken Language Processing.* Pittsburgh: ISCA, 2006. 273-276.

Bush, Michael, Alan K. Melby, Thor Anderson, Jeremy Browne, and Merrill Hansen. "Customized Video Playback: Standards for Content Modeling and Personalization." *Educational Technology* 44 (2004): 5-13.

Cautrell, Annie. "ESL Vs. EFL: What's the Difference?" *Associated Content.* August 02, 2007. http://www.associatedcontent.com/article/334179/esl_vs_efl_whats_the_difference.html?cat=4 (accessed 08 01, 2010).

EFR Team, Brigham Young University. "The EFR Project." *The EFR Project Web Page.* May 17, 2006. http://efr.byu.edu/ (accessed 08 01, 2010).

Jang, Photina Jaeyun, and Alexander G. Hauptmann. "Improving acoustic models with captioned multimedia speech." *IEEE International Conference on Multimedia Computing and Systems.* Florence: Carnegie Mellon University, 1999. 767-771.

Keeler, Farrah D.B. *Developing an Electronic Film Review for October Sky. Unpublished master's thesis.* Provo, Utah: Brigham Young University, 2005.

Kramsch, Claire, and Roger W. Andersen. "Teaching Text and Context Through Multimedia." *Language Learning & Technology* 2, no. 2 (January 1999): 31-42.

Li, Huiping, David Doermann, and Omid Kia. "Automatic text detection and tracking in digital video." *IEEE Transactions on Image Processing* 9, no. 1 (2000): 147-156.

Lin, Tong, and Hong-Jiang Zhang. "Automatic video scene extraction by shot grouping." *15th International Conference on Pattern Recognition (ICPR'00).* Barcelona: IEEE Computer Society, 2000. 4039.

LISA. *Term Base eXchange (TBX).* Romainmôtier: Localization Industry Standards Association, 2008.

Melby, Alan K. "TBX-Basic Translation-Oriented Terminology Made Simple." *Revista Tradumàtica*, 2009.

Melby, Alan K., Paul J. Fields, and Marc Carmen. "Language Databases, Statistics and Social Networks." *LACUS Forum XXXII.* Hanover: Dartmouth College, 2006. 133-143.

Melby, Alan Kenneth. "The EFR (Electronic Film Review) approach to using video in education." *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications.* 2004. 593-597.

Mora, Jill Kerper. "Methods of Second-Language Teaching." *MoraModules.* July 25, 1998. http://www.moramodules.com/L2MethodsMMdl/Default.htm (accessed June 25, 2010).

Och, Franz Josef, and Hermann Ney. "A Systematic Comparison of Various Statistical Alignment Models." *Computational Linguistics* 29 (March 2003): 19-51.

Secules, Teresa, Carol Herron, and Michael Tomasello. "The Effect of Video Context on Foreign Language Learning." *The Modern Language Journal* 76, no. 4 (1992): 480-490.

Tiedemann, Jörg. "News from OPUS - A Collection of Multilingual Parallel Corpora with Tools and Interfaces." In *Recent Advances in Natural Language Processing*, by N. Nicolov, K. Bontcheva, G. Angelova and R. Mitkov, 237-248. Amsterdam: John Benjamins, 2009.

Zhu, Weiyu, Candemir Toklu, and Shih-Ping Liou. "Automatic news video segmentation and categorization based on closed-captioned text." *2001 IEEE International Conference on Multimedia and Expo (ICME'01).* Tokyo: IEEE Computer Society, 2001. 211.

APPENDIX I: SOURCE CODE

The full Python 2.4 source code is included here for reference. The most current version of the software will also be available for electronic download at http://www.ttt.org/efr and http://buyog.com/efr.

A1.1. XCES2SEG: convert OPUS-style XCES script alignment files to EFR's Segfile format

```
#!/usr/bin/env python
"""XCES2Seg: processes an XCES file and produces a parallel file in EFR segmentation
format.

Usage: xces2seg.py -i [XCES file] -o [EFR segfile]
"""

import sys
import codecs
import os
import re
import getopt
import xml.etree.ElementTree as et

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "i:,o:,v:,t:,l:,d:")
    except getopt.error, msg:
        print __doc__
        print "\nParameter error(s):"
        print msg
        sys.exit(1)

    if opts:
        # get user-defined params in a more useful format
        params = opts_to_dict(opts)

        # ensure all required params have been provided and are valid
        validation_errors = validate_params(params)

        # if any (required) parameters are missing, terminate and report
        if validation_errors:
            print __doc__
            print "\nValidation error(s):"
            print '\n'.join(validation_errors)
            return 1

        # validation passed; proceed
        try:
            process(params)
        except Exception, e:
            print "Process failed!\n\nDetails: %s" % e

    else:
        print __doc__


def opts_to_dict(opts):
    """Takes an array of tuples, as returned by getopt,
```

```
       and returns a dict of corresponding name-value pairs"""
    params = {
        "xces_file": "",
        "seg_file": ""
    }
    for opt in opts:
        if opt[0] == "-i":
            params['xces_file'] = opt[1]
        elif opt[0] == "-o":
            params['seg_file'] = opt[1]
        elif opt[0] == "-v":
            params['dvdid'] = opt[1]
        elif opt[0] == "-t":
            params['title'] = opt[1]
        elif opt[0] == "-l":
            params['lang'] = opt[1]
        elif opt[0] == "-d":
            params['domain'] = opt[1]

    return params


def validate_params(params):
    """Takes a dict of name-value pairs representing command-line params,
       and validates that all required params are present and valid.
       Returns an array of error strings"""
    errors = []

    # check xces file
    if not params['xces_file']:
        errors.append("Missing XCES input file")
    elif not os.path.exists(params['xces_file']):
        errors.append("Specified XCES input file not found")

    # check segfile
    if not params['seg_file']:
        errors.append("Missing EFR segmentation file")
    elif not os.path.exists(params['seg_file']):
        errors.append("Specified EFR segmentation file not found")

    return errors


def process(params):
    """Main program loop; takes parameter dictionary as its sole input,
    and returns 0 if successful."""

    # load & parse the XCES
    raw_xml = et.parse(params['xces_file']).getroot()
    sentences = raw_xml.findall("s")   # get all sentences

    # write segfile header
    fOut = codecs.open(params['seg_file'], 'w', 'utf-16')
    fOut.write("//MediaID:%s\n" % params.get('dvdid','nnnnnnnnnnnnnnn'))
    fOut.write("//Title:%s\n" % params.get('title','1'))
    fOut.write("//Language:%s\n" % params.get('lang','en-US'))
    fOut.write("//subjectField:%s\n" % params.get('domain','General'))

    # for each sentence in the XCES:
    itr = sentences.__iter__()
    try:
        count = 0
        while 1:
```

```
            time0 = time1 = id = None
            allText = ''
            while not time1:
                t0, t1, id, words = getSentence(itr.next())
                if t0:
                    time0 = t0
                if t1:
                    time1 = t1
                allText = ' '.join([allText,words])

            smpte0 = timeToSmpte(time0)
            smpte1 = timeToSmpte(time1)
            f0 = smpteToFrameCode(smpte0)
            f1 = smpteToFrameCode(smpte1)
            fOut.write("%d-%d (%s - %s)\n" % (f0, f1, smpte0, smpte1))
            fOut.write("Scene %s\n" % id)
            fOut.write("//T:%s\n\n" % allText)
            count += 1

    except StopIteration:
        print "Found %d segments." % count

    except:
        print "Error while parsing XCES input:\n", sys.exc_info()

    fOut.close()
    return 0

def getSentence(s):
    # start <time>
    time0 = time1 = id = None
    if s[0].tag == 'time':
        time0 = s[0].get('value','00:00:00,00')
        id = s[0].get('id','TS')[1:-1]

    # child <w>ords
    words = s.findall("w")
    alltext = ' '.join([w.text for w in words])

    # end <time>
    if s[-1].tag == 'time':
        time1 = s[-1].get('value','00:00:00,00')

    return time0, time1, id, alltext

def smpteToFrameCode(smpte):
    t = smpte.split(':')
    h = int(t[0]) * 108000
    m = int(t[1]) * 1800
    s = int(t[2]) * 30
    f = int(t[3])
    return h+m+s+f

def timeToSmpte(timeStr):
    if timeStr.find(',') > -1:
        t = timeStr.split(',')
        return "%s:%d" % (t[0], int(t[1])*3/100)
    else:
        return timeStr

if __name__ == '__main__':
    main()
```

A1.2. CSV2MRC: convert CSV-style tabular data into the MRC TermTable format (which can then be transformed into TBX; see http://www.ttt.org/tbx/ for tools and documentation)

```
#!/usr/bin/env python
"""Produce MRC termbase files from a basic 4-column format
(subjectField,partOfSpeech,srcLang,tgtLang)

Usage: csv2mrc.py [input_file] [output_file]
"""

import sys
import os
import time


logLevel = ['Info','Warning','Error']
def _log(f, lvl, msg):
    f.write("%s [%s] %s\n" % (time.strftime("%H:%M:%S"), logLevel[lvl], msg))

def main():
    if len(sys.argv) > 2:
        args = sys.argv[1:]
    else:
        print __doc__,
        return 1

    if os.path.exists(args[0]):
        try:
            fIn = open(args[0], 'r')
            fOut = open(args[1], 'w')
            fLogOut = open(args[0]+'.log', 'w')

            srcText = tgtText = srcContext = tgtContext = ""
            iCount = 0

            # parse header line
            csvhead = fIn.readline()
            columns = csvhead.strip().split('\t')
            writeHeader(fOut, columns[2])

            while 1:
                line = fIn.readline()
                if not line: break

                #fLogOut.write("-> got line: '%s'\n" % line.strip())

                if not line.strip():
                    #ignore blank lines
                    pass

                else:
                    #_log(fLogOut,1,"Unmatched line: '%s'" % line)
                    cells = line.strip().split('\t')
                    iCount += 1

                    conceptId = "C%03d" % iCount
```

```
                    fOut.write("%s\tsubjectField\t%s\n" % (conceptId,cells[0]))
                    fOut.write("%s%s1\tterm\t%s\n" % (conceptId,columns[2],cells[2]))
                    fOut.write("%s%s1\tpartOfSpeech\t%s\n" %
(conceptId,columns[2],cells[1]))
                    fOut.write("%s%s1\tterm\t%s\n" % (conceptId,columns[3],cells[3]))
                    fOut.write("%s%s1\tpartOfSpeech\t%s\n" %
(conceptId,columns[3],cells[1]))

                    fOut.write('\n')

        except:
            print "offending line:'%s' (len:'%d')" % (line,len(line))
            print sys.exc_info()

        # write any end matter
        writeRespPerson(fOut)

        fIn.close()
        fOut.close()

def writeHeader(file, srcLang):
    file.write("=MRCtermTable\n")
    file.write("A\tworkingLanguage\t%s\n" % srcLang)
    file.write("A\tsourceDesc\tOpenSubs subtitle translations\n")

def writeRespPerson(file):
    file.write('\n')
    file.write("R001\ttype\tperson\n")
    file.write("R001\tfn\tRyan Corradini\n")
    file.write("R001\temail\tryancorradini@yahoo.com\n")
    file.write("R001\ttitle\tEFR software engineer\n")

if __name__ == '__main__':
    main()
```

A1.3. TBX PARSER: helper class, used by Linguassist (see A1.4) and MNF2VocTable (see A1.8).

```
#!/usr/bin/env python
"""Parses TBX file and provides term-lookup method to calling code.

Usage: parser.py [TBX file]
"""

import sys
import os
import re
import xml.etree.ElementTree as et

# XML namespace
__ns = 'http://www.w3.org/XML/1998/namespace'
__ns_esc = '{'+__ns+'}'

def parse(filename, mnf=False):
    termEntries = {}; indices = {}
    if os.path.exists(filename):
        try:
            raw_xml = et.parse(filename).getroot()
```

```
            entries = raw_xml.findall("text/body/termEntry")
            print "Got %d term entries." % len(entries)

            for entry in entries:
                id = entry.get("id")
                if id:
                    if not id in termEntries:
                        entryInfo = getTermEntryInfo(entry)
                        if entryInfo:
                            termEntries[id] = entryInfo
                            for lang in entryInfo['langSets']:
                                ls = entryInfo['langSets'][lang]
                                for tigID in ls['tigs']:
                                    tig = ls['tigs'][tigID]
                                    langCode = ls['lang']
                                    if not langCode in indices:
                                        indices[langCode] = {}
                                    indices[langCode][tig['term'].lower()] = id
                    else:
                        print "Duplicate termEntry id:", id
                else:
                    print "No termEntry id; skipping this entry"

        except:
            print "Error processing TBX:\n", sys.exc_info()
            return None, None

    return (termEntries, indices)

def getTermEntryInfo(termEntryNode):
    entryInfo = {'id': termEntryNode.get("id"),
                 'term_index':{},
                 'langSets':{},
                 'metadata':[]}
    currLangSet = None
    currGrp = None
    currTig = None
    currLevel = entryInfo

    itr = termEntryNode.getiterator()
    for el in itr[1:]: # ignore the termEntry node
        if el.tag == "langSet":
            # close out the current langSet, if there is one, and create a new one
            if currLangSet:
                if currLevel == currTig:
                    if currGrp:
                        currTig['metadata'].append(currGrp)
                        currGrp = None
                    currLangSet['tigs'][currTig['id']] = currTig
                    currTig = None
                if currGrp:
                    currLangSet['metadata'].append(currGrp)
                    currGrp = None

                entryInfo['langSets'][currLangSet['lang']] = currLangSet
            currLangSet = {"lang":el.get(__ns_esc+"lang"),'metadata':[],'tigs':{}}
            currLevel = currLangSet

        elif el.tag[-3:] == "Grp":
            # nested element
            if currGrp:
                currLevel['metadata'].append(currGrp)
            currGrp = {}
```

```
        elif el.tag[-3:] == 'tig':
            if currTig:
                currLangSet['tigs'][currTig['id']] = currTig
                currTig = None
            currTig = {"id": el.get('id'),'metadata':[],'term':None}
            currLevel = currTig

        elif el.tag == 'term':
            if not currTig:
                if 'term' in currLevel:
                    currTig = currLevel
                else:
                    currTig = {'id':None,'term':None,'metadata':[]}
            currTig['term'] = el.text
            entryInfo['term_index'][currLangSet['lang']] = el.text

        elif 'type' in el.keys():
            # basic element
            if currLevel:
                currLevel['metadata'].append({el.get('type'):el.text})

        else:
            # everything is attached to the term entry itself
            print "unhandled tag:",el

        # close out the last remaining langSet, Group, whatever
        if currLangSet:
            if currLevel == currTig:
                if currGrp:
                    currTig['metadata'].append(currGrp)
                    currGrp = None
                currLangSet['tigs'][currTig['id']] = currTig
                currTig = None
            if currGrp:
                currLangSet['metadata'].append(currGrp)
                currGrp = None

            entryInfo['langSets'][currLangSet['lang']] = currLangSet

    return entryInfo

if __name__ == '__main__':
    if len(sys.argv) > 2:
        args = sys.argv[1:]
    else:
        print __doc__,
        sys.exit(1)

    parse(args[0])
```

A1.4. EFR SEGFILE PARSER: Helper module for parsing EFR segfiles; used by A.1.5, Linguassist.

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

import os
import re
```

```
sTimecodeMatch = u"[0-9]{2}:[0-9]{2}:[0-9]{2}:[0-9]{2}"
reSegComment = re.compile(u"^//([a-z]+):(.*)$", re.I)
reSegHeader = re.compile(u"^[0-9]+-[0-9]+[ ]+\(%s[ ]*-[ ]*%s\)$" % (sTimecodeMatch,
sTimecodeMatch))

def parse(filename):
    "Reads in an EFR segmentation file and returns a list of the marked-up terms of
interest."
    doc_attrs = {}
    current_segment = {}
    target_terms = []
    segments = []
    in_header = True

    if os.path.exists(filename):
        fIn = open(filename, 'r')

        while 1:
            line = fIn.readline()
            if not line: break
            line = line.strip() # eliminate trailing whitespace

            if reSegComment.match(line):
                matches = reSegComment.findall(line)[0]
                if len(matches) >= 2:
                    if in_header:
                        # save whatever this is as document metadata
                        doc_attrs[matches[0]] = matches[1]

                    else:
                        # save segment metadata
                        current_segment[u'attrs'][matches[0]] = matches[1]

                        # if this is a Transcript attribute, look for marked-up terms
of interest
                        if matches[0] == u"T":
                            termStart = matches[1].find(u"<item>")
                            termEnd = matches[1].find(u"</item>")
                            if termStart >= 0 and termEnd >= termStart:
                                term = matches[1][termStart+6:termEnd].strip()
                                target_terms.append(term)
                                current_segment[u'target_terms'].append(term)

            elif reSegHeader.match(line):
                #print "... segment header"
                if in_header: in_header = False
                try:
                    if current_segment:
                        segments.append(current_segment)
                    current_segment = {u'timecode':line, u'attrs':{}, u'target_terms':
[]}

                except:
                    print "bad line: %s\n(Exception: %s)\n" % (line,
str(sys.exc_info()))

            elif line:
                if current_segment:
                    current_segment['title'] = line

            else:
                # whitespace-only line: signals the end of the segment
```

```
                pass

        fIn.close()

    else:
        pass

    return doc_attrs, segments, target_terms

if __name__ == '__main__':
    import sys
    if len(sys.argv) >= 2:
        metadata, segs, terms = parse(sys.argv[1])
        print "Doc metadata:", metadata
        print "Terms of interest:", terms
```

A1.5. LINGUASSIST: Processes an EFR segfile, identifying all marked-up lexical items of interest and looking them up in the provided TBX termbase. Any items that are found are written out to a "hits" flat file, and any that are not found are output to a "missed" file, to be processed by Moses.

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

"""LinguAssist: processes an EFR segmentation file, looking for matching terminology
in
the provided TBX termbase, and outputting two lists: a 'hit-list' terms that were
found in the termbase, and a 'miss-list' of terms that were not found, prepared
for input to the Moses machine translation system.

Usage: linguassist.py -s [segmentation file] -t [TBX termbase]
                      -l [target lang] -h [hit-list filename] -m [miss-list filename]
"""

import sys
import codecs
import os
import re
import getopt
import segfile.parser as segparser
import TBX.parser as tbxparser

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "s:,t:,l:,h:,m:")
    except getopt.error, msg:
        print __doc__
        print "\nParameter error(s):"
        print msg
        sys.exit(1)

    if opts:
        # get user-defined params in a more useful format
        params = opts_to_dict(opts)

        # ensure all required params have been provided and are valid
```

```
        validation_errors = validate_params(params)

        # if any (required) parameters are missing, terminate and report
        if validation_errors:
            print __doc__
            print "\nValidation error(s):"
            print '\n'.join(validation_errors)
            return 1

        # validation passed; proceed
        try:
            process(params)
        except Exception, e:
            print "Process failed!\n\nDetails: %s" % e

    else:
        print __doc__


def opts_to_dict(opts):
    """Takes an array of tuples, as returned by getopt,
       and returns a dict of corresponding name-value pairs"""
    params = {
        "seg_file": "",
        "tbx_file": "",
        "tgt_lang": "",
        "out_hits": "",
        "out_missed": ""
    }
    for opt in opts:
        if opt[0] == "-s":
            params['seg_file'] = opt[1]
        elif opt[0] == "-t":
            params['tbx_file'] = opt[1]
        elif opt[0] == "-l":
            params['tgt_lang'] = opt[1]
        elif opt[0] == "-h":
            params['out_hits'] = opt[1]
        elif opt[0] == "-m":
            params['out_missed'] = opt[1]

    return params


def validate_params(params):
    """Takes a dict of name-value pairs representing command-line params,
       and validates that all required params are present and valid.
       Returns an array of error strings"""
    errors = []

    # check segfile
    if not params['seg_file']:
        errors.append("Missing EFR segmentation file")
    elif not os.path.exists(params['seg_file']):
        errors.append("Specified EFR segmentation file not found")

    # check TBX file
    if not params['tbx_file']:
        errors.append("Missing TBX termbase file")
    elif not os.path.exists(params['tbx_file']):
        errors.append("Specified TBX termbase file not found")

    # check target language
```

```
    if not params['tgt_lang']:
        errors.append("Missing target language")

    # check hits output file
    if not params['out_hits']:
        errors.append("Missing TBX hit-list file")

    # check MNF misses output file
    if not params['out_missed']:
        errors.append("Missing TBX miss-list file")

    return errors


def process(params):
    """Main program loop; takes parameter dictionary as its sole input,
    and returns 0 if successful."""

    print "--------------------\nRunning linguassist..."

    # load & index TBX
    print "\nParsing TBX:", params['tbx_file']
    entries, indices = tbxparser.parse(params['tbx_file'])
    #print entries['C070']
    #print indices['en']
    print "Done parsing TBX."

    # parse the segfile
    print "\nParsing segfile:", params['seg_file']
    metadata, segs, terms = segparser.parse(params['seg_file'])
    print "Processed %d segments, found %d lexical items of interest" %
(len(segs),len(terms))

    # custom subject field?
    srcLang = metadata.get('Language','en')[:2]
    tgtLang = params['tgt_lang'][:2]
    print "src/tgt languages:", srcLang, tgtLang
    subject = metadata.get("subjectField","")
    print "Custom subject field: %s" % subject

    # for each segment in the segfile:
    print "\nProcessing user-specified target terms..."
    fOutCsv = codecs.open(params['out_missed'], 'w', 'utf-16')
    fOutMnf = codecs.open(params['out_hits'], 'w', 'utf-16')
    for segment in segs:
        # for each target term:
        for term in segment['target_terms']:
            normalized = term.lower()
            #print "target term:", normalized
            # look up term+subjectField in TBX
            try:
                srcTag = "<src>%s</src>" % segment.get('title','')
                if normalized in indices[srcLang]:
                    # if found, save reference to target term
                    termEntry = entries[indices[srcLang][normalized]]
                    tgtTerm = termEntry['term_index'][tgtLang]
                    targetTerm = u"<item>%s</item>" % tgtTerm
                    fOutMnf.write("%s %s\n" % (targetTerm,srcTag))
                else:
                    # add to list for Moses input
                    #print "Term not found in source index. Marking for submission to
SMT."
                    termContext = segment[u'attrs'][u'T'] + ' ' + srcTag
```

```
                    fOutCsv.write(unicode(termContext) + '\n')
            except:
                print sys.exc_info()

    fOutMnf.close()
    fOutCsv.close()

    print "\nDone."

if __name__ == '__main__':
    main()
```

A1.6. OPUS2XLIFF: takes a plain-text OPUS sentence alignment file and converts it into XLIFF

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

"""Produce an XLIFF document from an OpenSubtitles transcript alignment file.

Usage: xliff.py -a [opus2xliff | xliff2moses]
                -s [source_lang]
                -t [target_lang]
                [input_file]
                [output filename]
"""

import sys
import os
import re
import getopt
import time
import xml.etree.ElementTree as et

# XLIFF namespace
__ns = 'urn:oasis:names:tc:xliff:document:1.2'
__ns_esc = '{'+__ns+'}'

logLevel = ['Info','Warning','Error']
def _log(f, lvl, msg):
    f.write("%s [%s] %s\n" % (time.strftime("%H:%M:%S"), logLevel[lvl], msg))

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "a:,s:,t:")

    except getopt.error, msg:
        print __doc__
        print "\nParameter error(s):"
        print msg
        return 1

    #print args, opts

    if opts:
        # get user-defined params in a more useful format
        params = opts_to_dict(opts)

        # validation passed; proceed
```

```
        try:
            # ensure all required params have been provided and are valid
            if validate_params(args, params):
                if params['action'] == 'opus2xliff':
                    opus2xliff(args[0], args[1], params['src_lang'],
params['tgt_lang'])

                elif params['action'] == 'xliff2moses':
                    xliff2moses(args[0], args[1])

                else:
                    print __doc__
                    print "Invalid action: %s" % params['action']
                    return 1

            else:
                return 1

        except Exception, e:
            print "Process failed!\n\nDetails: %s" % e

    else:
        print __doc__

def opts_to_dict(opts):
    """Takes an array of tuples, as returned by getopt,
       and returns a dict of corresponding name-value pairs"""
    params = {
        "action": "",
        "src_lang": "",
        "tgt_lang": ""
    }
    for opt in opts:
        if opt[0] == "-a":
            params['action'] = opt[1]
        elif opt[0] == "-s":
            params['src_lang'] = opt[1]
        elif opt[0] == "-t":
            params['tgt_lang'] = opt[1]

    return params


def validate_params(args, opts):
    """Takes a dict of name-value pairs representing command-line params,
       and validates that all required params are present and valid.
       Returns an array of error strings"""
    errors = []

    # check action
    if not opts['action']:
        errors.append("Missing action")

    # only require src/tgt if in Opus2XLIFF mode
    if opts['action'] == 'opus2xliff':
        # check source language
        if not opts['src_lang']:
            errors.append("Missing source language")

        # check target language
        if not opts['tgt_lang']:
            errors.append("Missing target language")
```

```
    # check input file
    if len(args) < 1:
        errors.append("Missing input file")
    elif not os.path.exists(args[0]):
        errors.append("Specified input file not found")

    # check output file
    if len(args) < 2:
        errors.append("Missing output filename")

    # if any (required) parameters are missing, terminate and report
    if errors:
        print __doc__
        print "\nValidation error(s):"
        print '\n'.join(errors)
        return False

    return True


def opus2xliff(in_file, out_file, src_lang, tgt_lang):
    bndRex = re.compile('^=+$')
    srcRex = re.compile('^\(src\)="([0-9]+)">(.*)')
    tgtRex = re.compile('^\(trg\)="([0-9]+)">(.*)')

    if os.path.exists(in_file):
        fIn = open(in_file, 'r')
        fOut = open(out_file, 'w')
        fLog = open('opus2xliff.log', 'w')

        # write header
        fOut.write("<xliff version='1.2' xmlns='%s'>\n" % __ns)
        fOut.write("<file original='%s' datatype='plaintext' \n      source-
language='%s' target-language='%s'>\n" % (in_file, src_lang, tgt_lang))
        fOut.write("<body>\n\t<group>\n")

        srcText = tgtText = srcContext = tgtContext = ""

        while 1:
            line = fIn.readline()
            if not line: break

            #fLog.write("-> got line: '%s'\n" % line.strip())

            if not line.strip():
                # blank lines mean the end of a film... clear our contexts
                srcText = tgtText = ""

            elif line[0] == '#':  # e.g. '# xml/eng'
                # lines starting with # indicate new src/tgt filenames (and new group)
                if srcContext+tgtContext:
                    srcContext = tgtContext = ""
                    fOut.write("\t</group>\n<group>\n")

            elif bndRex.match(line):
                # new segment... flush whatever is in our segment buffers
                if srcText+tgtText:
                    if srcText and tgtText:
                        fOut.write("\t\t<trans-unit id='%s'>\n" % srcContext)
                        fOut.write("\t\t\t<source>%s</source>\n" % srcText)
                        fOut.write("\t\t\t<target>%s</target>\n" % tgtText)
                        fOut.write("\t\t</trans-unit>\n")
```

```
                    else:
                        msg = "Ignoring incomplete segment (src:'%s'; tgt:'%s')" %
(srcText, tgtText)
                        _log(fLog,1,msg)

                else:
                    # In practice, these "empty segments" are really just an
                    # artifact of the parsing process at the head of each film.
                    # --> Safe to ignore.
                    pass

                srcText = tgtText = ""

            elif srcRex.match(line):
                srcContext = srcRex.findall(line)[0][0]
                srcText += cleanXml( srcRex.findall(line)[0][1].strip() )

            elif tgtRex.match(line):
                tgtContext = tgtRex.findall(line)[0][0]
                tgtText += cleanXml( tgtRex.findall(line)[0][1].strip() )

            else:
                _log(fLog, 1,"Unmatched line: '%s'" % line)

        # write footer
        fOut.write("\t</group>\n</body>\n</file>\n</xliff>")

        print "Wrote output file %s" % out_file

        fIn.close()
        fOut.close()

    else:
        print "Error: Specified OpenSubs alignment file not found"

    pass

def xliff2moses(in_file, out_file_base):
    if os.path.exists(in_file):
        try:
            # parse input XLIFF
            et._namespace_map['xl'] = __ns
            xliff = et.parse(in_file).getroot()

            # open log file
            fLog = open('xliff2moses.log', 'w')

            # grab source/target language codes from XLIFF, use them to create the
output files
            fileTag = xliff.find(__ns_esc+'file')
            if fileTag:
                srcLang = fileTag.get('source-language')
                tgtLang = fileTag.get('target-language')
            errors = []
            if not srcLang:
                errors.append('XLIFF document is missing source-language attribute')
            if not tgtLang:
                errors.append('XLIFF document is missing target-language attribute')
            if errors:
                raise Exception('\n'.join(errors))
        except:
            exc = sys.exc_info()
            print "Failed to parse the specified XML.\n Details: %s\nLine #: %d" %
```

```
(exc[1], exc[2].tb_lineno)


        # successfully parsed the input; now chew through it to produce the outputs
        print "source lang: '%s'; target lang: '%s'" % (srcLang, tgtLang)
        out_path1 = '%s.%s' % (out_file_base, srcLang)
        out_path2 = '%s.%s' % (out_file_base, tgtLang)
        out_file1 = open(out_path1, 'w')
        out_file2 = open(out_path2, 'w')

        # grab all source/target elements from the XLIFF, and output them to the
src/tgt files
        xpath = 'ns:body/ns:group/ns:trans-unit'.replace('ns:',__ns_esc)
        terms = fileTag.findall(xpath)
        print "Got %d grouped translation units" % len(terms)
        _write_terms(terms, out_file1, out_file2, fLog)

        # now do it again for any ungrouped trans-units
        xpath = 'ns:body/ns:trans-unit'.replace('ns:',__ns_esc)
        terms = fileTag.findall(xpath)
        print "Got %d ungrouped translation units" % len(terms)
        _write_terms(terms, out_file1, out_file2, fLog)

        print "Wrote %s and %s output files." % (out_path1, out_path2)

        out_file1.close()
        out_file2.close()

    else:
        print "Error: Specified OpenSubs alignment file not found"

    return

def _write_terms(terms, out1, out2, fLog):
    for trans_unit in terms:
        try:
            src = trans_unit.findall(__ns_esc+"source")
            tgt = trans_unit.findall(__ns_esc+"target")
            if src and tgt:
                for l in src: out1.write(l.text.encode("UTF-8"))
                out1.write('\n')

                for l in tgt: out2.write(l.text.encode("UTF-8"))
                out2.write('\n')
            else:
                _log(fLog,1,"missing src or tgt for this trans_unit (id: %s)" %
trans_unit.get("id"))
        except Exception, e:
            msg = "Skipped this trans-unit (id: %s) (error: %s)" %
(trans_unit.get("id"), e)
            _log(fLog,1,msg)

def cleanXml(content):
    temp = content.replace('&','&amp;')
    temp = temp.replace('>','&gt;')
    temp = temp.replace('<','&lt;')

    return temp

if __name__ == '__main__':
    main()
```

A1.7. XLIFF2MOSES: splits an XLIFF bitext into source/target language files to use in Moses training

(This routine is part of the same source code as A1.6, OPUS2XLIFF)

A1.8. MOSES2MNF: correlate Moses input and output files into a unified MNF file

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

"""Produce a TBX file in the MNF (Mapped iNterchange file) schema, given parallel
input and output files from Moses.

Usage: Moses2MNF.py -i [moses_input_file] -o [moses_output_file] -m [target_mnf_file]
                    -s [source lang] -t [target lang]
"""

import sys
import os
import re
import codecs
import getopt

TBXHeader = """<?xml version='1.0' encoding="UTF-16"?>
<!DOCTYPE martif SYSTEM "TBXcoreStructV02.dtd">
<martif type="TBX" xml:lang="en">
    <martifHeader>
        <fileDesc><sourceDesc><p>__FILENAME__</p></sourceDesc></fileDesc>
        <encodingDesc>
            <p
type="XCSURI">http://www.lisa.org/fileadmin/standards/tbx/TBXXCSV02.xcs</p>
        </encodingDesc>
    </martifHeader>
    <text>
        <body>"""

MNFSegmentRef = "<admin type='sourceSegment'>__SEGID__</admin>"
MNFEntry = """
<termEntry id="__ID__">
    <descrip type="subjectField">__SUBJECT_FIELD__</descrip>
    __SEGMENTS__
    <langSet xml:lang="__LANG0__">
        <tig>
            <term>__TERM0__</term>
            <descrip type="context">__CTX0__</descrip>
        </tig>
    </langSet>
    <langSet xml:lang="__LANG1__">
        <tig>
            <term>__TERM1__</term>
            <descrip type="context">__CTX1__</descrip>
        </tig>
    </langSet>
</termEntry>"""

TBXFooter = "</body></text></martif>"

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "i:,o:,m:,s:,t:")
```

```
    except getopt.error, msg:
        print __doc__
        print "\nParameter error(s):"
        print msg
        return 1

    if opts:
        # get user-defined params in a more useful format
        params = opts_to_dict(opts)

        # validation passed; proceed
        try:
            # ensure all required params have been provided and are valid
            if validate_params(args, params):
                moses2mnf(args,params)

            else:
                return 1

        except Exception, e:
            print "Process failed!\n\nDetails: %s" % e

    else:
        print __doc__

def opts_to_dict(opts):
    """Takes an array of tuples, as returned by getopt,
       and returns a dict of corresponding name-value pairs"""
    params = {
        "in_file": "",
        "out_file": "",
        "mnf_file": "",
        "src_lang": "",
        "tgt_lang": ""
    }
    for opt in opts:
        if opt[0] == "-i":
            params['in_file'] = opt[1]
        elif opt[0] == "-o":
            params['out_file'] = opt[1]
        elif opt[0] == "-m":
            params['mnf_file'] = opt[1]
        elif opt[0] == "-s":
            params['src_lang'] = opt[1]
        elif opt[0] == "-t":
            params['tgt_lang'] = opt[1]

    return params

def validate_params(args, opts):
    """Takes a dict of name-value pairs representing command-line params,
       and validates that all required params are present and valid.
       Returns an array of error strings"""
    errors = []

    # check Moses in-file
    if not opts['in_file']:
        errors.append("Missing Moses in-file")

    # check Moses out-file
    if not opts['out_file']:
        errors.append("Missing Moses out-file")
```

```
    # check Moses in-file
    if not opts['mnf_file']:
        errors.append("Missing target MNF filename")

    # check source lang
    if not opts['src_lang']:
        errors.append("Missing source language")

    # check target lang
    if not opts['tgt_lang']:
        errors.append("Missing target language")

    # if any (required) parameters are missing, terminate and report
    if errors:
        print __doc__
        print "\nValidation error(s):"
        print '\n'.join(errors)
        return False

    return True

def moses2mnf(args,params):
    fIn = codecs.open(params['in_file'],'r',encoding='utf-8')
    fOut = codecs.open(params['out_file'],'r',encoding='utf-8')
    fMnf = codecs.open(params['mnf_file'],'w',encoding='utf-16')

    newFilename = "%s/%s" % (params['in_file'],params['out_file'])
    fMnf.write(TBXHeader.replace("__FILENAME__",newFilename))

    while 1:
        lineIn = fIn.readline()
        lineOut = fOut.readline()
        if not lineIn or not lineOut: break
        lineIn = lineIn.strip() # eliminate trailing whitespace
        lineOut = lineOut.strip() # eliminate trailing whitespace

        # build the termEntry
        term0 = extractXmlContent(lineIn, "item").strip()
        term1 = extractXmlContent(lineOut, "item").strip()
        termEntry = MNFEntry.replace("__ID__", term0.replace(" ","_")+'.1')
        termEntry = termEntry.replace("__LANG0__", params['src_lang'])
        termEntry = termEntry.replace("__LANG1__", params['tgt_lang'])
        termEntry = termEntry.replace("__TERM0__", term0)
        termEntry = termEntry.replace("__CTX0__", stripXml(lineIn))
        termEntry = termEntry.replace("__TERM1__", term1)
        termEntry = termEntry.replace("__CTX1__", stripXml(lineOut))
        termEntry = termEntry.replace("__SUBJECT_FIELD__", "General")
        segments = ""
        for segId in extractXmlContent(lineIn, "src").split(','):
            segments = segments + MNFSegmentRef.replace("__SEGID__", segId) + '\n'
        termEntry = termEntry.replace("__SEGMENTS__", segments)

        fMnf.write(termEntry)

    fMnf.write(TBXFooter)

    fIn.close()
    fOut.close()
    fMnf.close()

    print "Done."
```

```
def extractSegs(line):
    pos0 = line.find("<src>")
    pos1 = line.find("</src>")
    src = line
    if pos0+pos1 > 0:
        src = line[pos0+6:pos1-pos0]
    return src

def extractXmlContent(text, tagName, includeTags=False):
    tagOpen = "<%s>" % tagName
    tagClose = "</%s>" % tagName
    tagOpenLen = len(tagOpen)
    tagCloseLen = len(tagClose)

    pos0 = text.find(tagOpen)
    pos1 = text.find(tagClose)
    if pos0+pos1 > 0:

        if includeTags:
            pos1 = pos1 + tagCloseLen
        else:
            pos0 = pos0 + tagOpenLen

        content = text[pos0:pos1]

    else:
        content = text

    return content

def stripXml(line):
    # find <item>
    pos0 = line.find("<item>")
    pos1 = line.find("</item>")

    output = line[:pos0] + line[pos0+6:pos1].strip() + line[pos1+7:]

    # find <src>
    pos0 = output.find("<src>")
    pos1 = output.find("</src>")
    output = output[:pos0] + output[pos1+6:]

    return output

def cleanXml(content):
    temp = content.replace('&','&amp;')
    temp = temp.replace('>','&gt;')
    temp = temp.replace('<','&lt;')

    return temp


if __name__ == '__main__':
    main()
```

A1.9. MNF2VOCTABLE: takes the formatted term-lookup and Moses suggestion list and writes the combined results to an EFR VocTable (in CSV format)

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

"""Produce an EFR VocTable in CSV format, given a Moses-originated MNF(TBX) file.

Usage: mnf2voctable.py -i [input_mnf] -o [output_csv] -s [source_lang] -t
[target_lang]
"""

import codecs
import sys
import os
import re
import getopt
import time
import parser as tbx_parser


csvColumns = ['Chapter #','Token #','Token','Utterance',
              'Concept ID','Head Word','Grammar',
              'Definition','Examples','Optional Notes',
              'Translation','Level','Images']
logLevel = ['Info','Warning','Error']
def _log(f, lvl, msg):
    f.write("%s [%s] %s\n" % (time.strftime("%H:%M:%S"), logLevel[lvl], msg))

def main():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "i:,o:,s:,t:")

    except getopt.error, msg:
        print __doc__
        print "\nParameter error(s):"
        print msg
        return 1

    #print args, opts

    if opts:
        # get user-defined params in a more useful format
        params = opts_to_dict(opts)

        # validation passed; proceed
        try:
            # ensure all required params have been provided and are valid
            if validate_params(args, params):
                fIn = codecs.open(params['in_file'], 'r')
                fOut = codecs.open(params['out_file'], 'w', encoding='utf-16')
                lang0 = params['src_lang']
                lang1 = params['tgt_lang']

                # write header
                sHeader = unicode(','.join(csvColumns) + '\n')
                fOut.write(sHeader)

                # parse MNF
                (termEntries, indices) = tbx_parser.parse(params['in_file'], mnf=True)

                # write glossary rows, followed by all subsequent references to those
rows
                tokenCount = 0
                for entryID in termEntries:
                    tokenCount = tokenCount + 1
                    entry = termEntries[entryID]
```

```
                    term0 = entry['terms'][lang0]
                    term1 = entry['terms'][lang1]
                    token = csvEscape(term0.split()[0])
                    for i in range(0,len(entry['sourceSegments'])):
                        rowData = ['1', str(tokenCount), token,
entry['sourceSegments'][i], entryID]
                        if i==0:
                            # on first instance, we include all the row data
                            termEntry = [
                                csvEscape(term0),  # head word
                                '',     # grammar
                                '',     # definition
                                csvEscape(entry['context'][lang0]),    # examples
                                '',     # notes
                                csvEscape(term1),  # translation
                                '100',    # level
                                ''      # image
                            ]
                            rowData.extend(termEntry)

                        else:
                            # on all subsequent instances, just output columns 1-5
                            # (i.e. add 9 empty cols)
                            for i in range(9): rowData.append('')

                        sRow = unicode(','.join(rowData) + '\n')
                        fOut.write(sRow)


            else:
                return 1

        except Exception, e:
            print "Process failed!\n\nDetails: %s" % e

    else:
        print __doc__

def opts_to_dict(opts):
    """Takes an array of tuples, as returned by getopt,
       and returns a dict of corresponding name-value pairs"""
    params = {
        "in_file": "",
        "out_file": "",
        "src_lang": "",
        "tgt_lang": ""
    }
    for opt in opts:
        if opt[0] == "-i":
            params['in_file'] = opt[1]
        elif opt[0] == "-o":
            params['out_file'] = opt[1]
        elif opt[0] == "-s":
            params['src_lang'] = opt[1]
        elif opt[0] == "-t":
            params['tgt_lang'] = opt[1]

    return params


def validate_params(args, opts):
    """Takes a dict of name-value pairs representing command-line params,
       and validates that all required params are present and valid.
```

```
       Returns an array of error strings"""
    errors = []

    # check input file
    if not opts['in_file']:
        errors.append("Missing input file (-i)")
    elif not os.path.exists(opts['in_file']):
        errors.append("Specified input file not found")

    # check output file
    if not opts['out_file']:
        errors.append("Missing output filename (-o)")

    # check source lang
    if not opts['src_lang']:
        errors.append("Missing source language (-s)")

    # check output file
    if not opts['tgt_lang']:
        errors.append("Missing target language (-t)")

    # if any (required) parameters are missing, terminate and report
    if errors:
        print __doc__
        print "\nValidation error(s):"
        print '\n'.join(errors)
        return False

    return True


def csvEscape(text):
    if text.find(',') >= 0:
        return '"%s"' % text
    else:
        return text

if __name__ == '__main__':
    main()
```

Appendix II: Getting Moses Working on Cygwin

The Moses decoder and its associated tools are primarily intended to be used on UNIX platforms, but have stated compatibility with Cygwin as well. However, getting things to actually build on that platform isn't completely straightforward. This section provides some comments on that process.

A2.1. SUPPORT TOOLS. I set out following the "Installation and Training Run-Through" guidelines published on the official Moses web site (http://www.statmt.org/moses_steps.html), but the following modifications were needed:

GIZA++

- Make sure when you edit the Makefile, that your editor doesn't strip out the tabs and replace them with spaces. Many editors do this by default, but it will cause hard-to-identify errors in the code.
- A linker error of the form "`cannot find -lgcc_s`" indicates that you need to comment out the line that reads "`LDFLAGS = -static`", and instead use "`LDFLAGS =`" (this appears to be due to a lack of support for static links in Windows and/or Cygwin)

SRILM

- The machine-type script won't run without the tcsh shell installed within Cygwin; see here: http://www.speech.sri.com/pipermail/srilm-user/2010q2/000871.html
    - This is especially interesting in light of the comment on the Moses tutorial that "These instructions work on bash."

IRSTLM

- Running "`make install`" may give weird errors referring to a "NONE" folder. If so, open up the Makefile in the IRSTLM root, plus those in the `src`, `scripts`, and `example` subfolders, and replace all instances of "NONE" with ".."

A2.2. COMPILING MOSES. The Moses source code comes with project files for Eclipse, Xcode, and Visual Studio, but these are all largely unsupported and not recommended by the development team. However, using the Linux command-line method of compilation resulted in a much larger binary (36.7 megabytes, as opposed to the 504 kilobytes taken up by the Windows binary), which loaded and ran much more slowly than its native counterpart. Where possible, therefore, it is advisable to build and run the decoder in a native environment, rather than in Cygwin (The language modeling tools, however, can only be compiled and run in a Unix-like environment).

Another problem to bear in mind is that of version incompatibility. If you download an old binary of the Moses decoder, but compile the language modeling tools from source, you may encounter errors such as the following, which I saw when running a new language model with a 2-year-old Moses decoder:

```
Loading lexical distortion models...
have 1 models
Lexical model type underspecified for model 0!
```

When I compiled a clean copy of Moses from the latest source, this error went away.

A2.3. MOSES SUPPORT SCRIPTS. Moses includes a set of scripts for use in building language models, formatting training data, and so on. As before, I tried to follow the steps outlined by the setup guide, but encountered quite a few issues:

1. It seems that in certain instances, it may be necessary to enclose the TARGETDIR and BINDIR values in the makefile with quote characters, even though the versions in the sample comment aren't so delimited.

2. Even though the compiled binaries are correctly copied to $BINDIR, they have the wrong extension (Unix executables have no extension at all, but Windows/Cygwin binaries end in .exe); this leads to the following error:

```
Please specify a BINDIR.
The BINDIR directory must contain GIZA++, snt2cooc.out and mkcls executables.
These are available from http://www.fjoch.com/GIZA++.html  and
http://www-i6.informatik.rwth-aachen.de/Colleagues/och/software/mkcls.html.
```

The way to fix this is to edit `.\scripts\check-dependencies.pl` to comment out the pertinent block, or change the "unless" clause to include ".exe" in the filenames:

```
$ diff scripts/check-dependencies.pl scripts/check-dependencies.pl.unfixed
3c3
< # $Id: check-dependencies.pl 1331 2007-03-26 20:06:44Z hieuhoang1972 $
---
> # $Id: check-dependencies.pl.unfixed 1331 2007-03-26 20:06:44Z hieuhoang1972 $
28,37c28,37
< #unless (-x "$bin_dir/GIZA++" && -x "$bin_dir/snt2cooc.out" && -x "$bin_dir/mkcls" )
{
< #  print <<EOT;
< #Please specify a BINDIR.
< #
< #  The BINDIR directory must contain GIZA++, snt2cooc.out and mkcls executables.
< #  These are available from http://www.fjoch.com/GIZA++.html and
< #  http://www-i6.informatik.rwth-aachen.de/Colleagues/och/software/mkcls.html .
< #EOT
< #  exit 0;
< #}
---
> unless (-x "$bin_dir/GIZA++" && -x "$bin_dir/snt2cooc.out" && -x "$bin_dir/mkcls" )
{
>   print <<EOT;
> Please specify a BINDIR.
>
>   The BINDIR directory must contain GIZA++, snt2cooc.out and mkcls executables.
>   These are available from http://www.fjoch.com/GIZA++.html and
>   http://www-i6.informatik.rwth-aachen.de/Colleagues/och/software/mkcls.html .
> EOT
>   exit 0;
> }
```

3. `Make` tries to invoke `rsync`, which isn't installed by default in Cygwin. Re-run the Cygwin installer with Internet access to get the pertinent package.

4. Once `rsync` was installed, the "`make release`" process mostly worked–except for a few binaries it expected, but couldn't find:

```
/scripts/training/lexical-reordering/score.exe
```

```
/scripts/training/memscore/memscore.exe
/scripts/training/mbr/mbr.exe
rsync error: some files/attrs were not transferred (see previous errors) (code 23) at
/home/lapo/packaging/rsync-3.0.7-1/src/rsync-3.0.7/main.c(1042) [sender=3.0.7]
make: *** [release] Error 23
```

I noticed that my copy of the makefile didn't include those three, so I got a clean copy from the original package, and re-ran "`make release`". This time, `make` *failed* on the three missing binaries:

a. score.exe: build failed:

```
g++ -O6 -g -c reordering_classes.cpp
g++ -lz score.cpp reordering_classes.o -o score
reordering_classes.o:/cygdrive/c/LingMA/moses/scripts/training/lexical-
reordering/reordering_classes.cpp:348: undefined reference to `_gzopen'
reordering_classes.o:/cygdrive/c/LingMA/moses/scripts/training/lexical-
reordering/reordering_classes.cpp:352: undefined reference to `_gzwrite'
reordering_classes.o:/cygdrive/c/LingMA/moses/scripts/training/lexical-
reordering/reordering_classes.cpp:355: undefined reference to `_gzclose'
collect2: ld returned 1 exit status
make: *** [score] Error 1
```

It seems the Makefile is actually wrong; the -lz should actually be *after* the .cpp and .o files:

```
diff /scripts/training/lexical-reordering/Makefile /scripts/training/lexical-
reordering/Makefile.old
11c11
<   $(CXX) -lz score.cpp reordering_classes.o -o score
---
>   $(CXX) score.cpp reordering_classes.o -lz -o score
```

b. memscore.exe: build failed, but according to this note in the makefile, it's not a problem:

```
# Building memscore may fail e.g. if boost is not available.
# We ignore this because traditional scoring will still work and memscore isn't used
by default.
```

c. mbr.exe: build failed:

```
g++ -O3    mbr.cpp   -o mbr
mbr.cpp: In function `int main(int, char**)':
mbr.cpp:367: error: `time' was not declared in this scope
make: *** [mbr] Error 1
```

To fix this error, we need to add an explicit reference to time.h:

```
$ diff scripts/training/mbr/mbr.cpp scripts/training/mbr/mbr.cpp.old
12d11
```

```
< #include <time.h>     /* added by RAC for compilation on Cygwin */
```

5. The Moses tutorial specifies a training script, `training/train-factored-phrase-model.perl`, which is not present in the latest package. However, looking at the file `training/train-model.perl`, a comment referred to it as "Train Factored Phrase Model", so apparently the name has just changed without the documentation being updated to reflect it.
6. I tried to run `training/train-model.perl`, following the example syntax from the tutorial, and got a "permission denied" error:

```
bash: tools/moses-scripts/scripts-20100605-2158/training/train-model.perl:
Permission denied
```

   This one is simple to fix. In Windows Explorer, right-click the `/moses/tools` folder and select "Properties". On the Security tab, click Advanced, then give your user full control over that folder, overriding any existing permissions on either the ancestors or descendants.

7. Once I had a complete scripts build, and had successfully invoked `train-model.perl` to build a language model and phrase model, the next step was to 'sanity check' the new models:

```
echo "c' est une petite maison ." | ./moses -f work/model/moses.ini
```

This failed with a large cascade of text, but buried within that stream is this error:

```
ERROR:File 5 does not exist
```

This apparently refers to the following section of the generated moses.ini file:

```
# translation tables: source-factors, target-factors, number of scores, file
[ttable-file]
0 0 0 5 /cygdrive/c/LingMA/moses/work/model/phrase-table.gz
```

The comment seems to indicate that the section will have 4 parameters (source-factors, target-factors, number of scores, and file), but the following line instead has 5 params (0, 0, 0, 5, and the path to `phrase-table.gz`); this tracks with Moses's complaint about not being able to find a *file* (parameter 4 in this section) named "5" (value 4 in that same section). I fixed this by removing one of the (apparently redundant) zeroes:

```
$ diff moses.ini moses.ini.old
15c15
< 0 0 5 ./model/phrase-table
---
> 0 0 0 5 ./model/phrase-table
```

Having made all of the above changes, I was able to successfully follow the rest of the Moses installation and training run-through.

A2.4. PYTHON UNICODE ISSUES. Even once you have succeeded in getting Moses and its support tools and scripts running under Cygwin, there is one more issue to be aware of regarding Cygwin Python's file commands. If directed to read and write multi-byte data of any kind (e.g. UTF-8, UTF-16, etc) to a file, Python will do so successfully. Files created using full 2-byte Unicode (referred to by Python as UTF-16) will receive the correct Byte-Order Mark (BOM) that will identify the file's encoding type to all other Windows applications. However, if creating a new UTF-8 file (which would be preferable in most circumstances due to its more efficient storage mechanism), the file doesn't receive the UTF-8 BOM… so Windows applications, with no evidence to the contrary, will detect the file encoding as ASCII. In practice this may not cause any problems, as long as the data is read and written solely by code that has a foreknowledge of its encoding type, and handles it as such. But it *is* an issue that must be remembered when dealing with international data.