PBML

The Prague Bulletin of Mathematical Linguistics
NUMBER 93 JANUARY 2010 117–126

The Machine Translation Toolpack for LoonyBin: Automated Management of Experimental Machine Translation HyperWorkflows

Jonathan H. Clark[a], Jonathan Weese[b], Byung Gyu Ahn[b], Andreas Zollmann[a], Qin Gao[a], Kenneth Heafield[a], Alon Lavie[a]

[a] Language Technologies Institute, Carnegie Mellon University
[b] Center for Language and Speech Processing, Johns Hopkins University

Abstract

Construction of machine translation systems has evolved into a multi-stage workflow involving many complicated dependencies. Many decoder distributions have addressed this by including monolithic training scripts – `train-factored-model.pl` for Moses and `mr_runmer.pl` for SAMT. However, such scripts can be tricky to modify for novel experiments and typically have limited support for the variety of job schedulers found on academic and commercial computer clusters. Further complicating these systems are hyperparameters, which often cannot be directly optimized by conventional methods requiring users to determine which combination of values is best via trial and error. The recently-released LoonyBin open-source workflow management tool addresses these issues by providing: 1) a visual interface for the user to create and modify workflows; 2) a well-defined logging mechanism; 3) a script generator that compiles visual workflows into shell scripts, and 4) the concept of Hyperworkflows, which intuitively and succinctly encodes small experimental variations within a larger workflow. In this paper, we describe the Machine Translation Toolpack for LoonyBin, which exposes state-of-the-art machine translation tools as drag-and-drop components within LoonyBin.

1. LoonyBin Background

Empirical research in machine translation has become a complex multi-stage process with many stages being run under multiple experimental conditions (i.e. with different corpora and different sets of hyperparameters). The management of such

Corresponding author: jhclark@cs.cmu.edu
Cite as: Jonathan H. Clark, Jonathan Weese, Byung Gyu Ahn, Andreas Zollmann, Qin Gao, Kenneth Heafield, Alon Lavie. The Machine Translation Toolpack for LoonyBin: Automated Management of Experimental Machine Translation HyperWorkflows. The Prague Bulletin of Mathematical Linguistics No. 93, 2010, pp. 117–126. ISBN 978-80-904175-4-0. doi: 10.2478/v10108-010-0002-x.

workflows presents a real challenge in terms of keeping results organized, analyzing results at every stage, and automating the workflow.

For example, in syntactic statistical machine translation, a typical experiment consists of over 20 tools with a complex network of dependencies spanning multiple machines or even clusters of machines. Parsing and phrase extraction might be run on a large cluster of hundreds of low-memory machines, preprocessing and word alignment might be run on a local server, while tuning and decoding might be done on a small cluster of large-memory machines. Further, this system might be run for two language pairs and using 10 sets of features in the translation model to verify some experimental hypothesis.

With these needs in mind, LoonyBin (Clark and Lavie, 2010) accommodates workflows that:

- span various machines, clusters, and schedulers
- involve many separate tools, which can be invoked by arbitrary UNIX commands
- have components that are run multiple times under multiple conditions
- evolve quickly with tools frequently being added, removed, and swapped

LoonyBin accomplishes this by providing the following advantages over current common practices:

- associating sanity checks and logging directly with tools, separating these from ad hoc wrappers and automation scripts
- maintaining a cleanly organized directory structure for each step and each condition under which a step is run
- providing a resume-on-failure mechanism for every stage in the pipeline
- making it easy for those without a detailed knowledge of each tool's internals to run the system by providing textual descriptions of each parameter, input file, and output file in a graphical workflow designer
- automatically copying required files between machines/clusters via SSH
- compiling workflows into shell scripts, a medium already in widespread use by NLP researchers

1.1. Workflow Semantics

We now discuss the representation of workflows in LoonyBin. In their most basic form, LoonyBin represents workflows as Directed Acyclic Graphs (DAGs). In this form, each vertex represents a TOOL, which produces output files given input files and parameters, and directed edges indicate relative temporal ordering of tools and information flow (files or parameters) by mapping the output of one tool to the inputs of the next. A TOOL DESCRIPTOR defines the commands necessary to run a tool given inputs, outputs, and parameters. Custom tool descriptors can be implemented via simple user-defined Python scripts that generate shell commands. These tool descriptors contain PRE-ANALYZERS to check the sanity of the inputs and log information

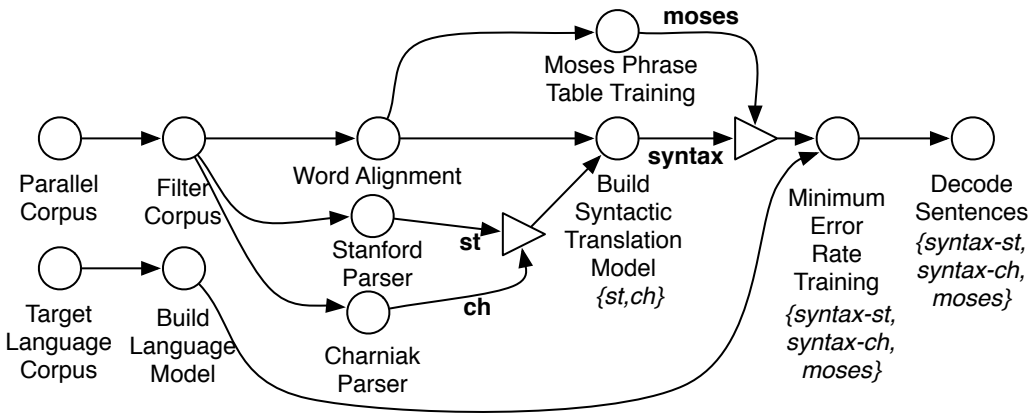

Figure 1. A simplified version of the CMU StatXfer system HyperWorkflow for the GALE Phase 4 Machine Translation Evaluation showing the multiple experiments that were run

and POST-ANALYZERS to check the sanity of the output files, log information about the outputs, and extract log data from any third-party log file formats.

1.2. HyperWorkflow Semantics

LoonyBin also represents the running of workflows under multiple experimental conditions (i.e. with different input files or parameters). We call this a HYPERWORKFLOW. A HyperWorkflow contains REALIZATION VARIABLES, which introduce variations into a shared workflow. Each realization variable can take on a REALIZATION VALUE, which is a set of files and parameters. For instance the realization variable "language model file and order" could take on the realization value {english.txt, 4}. Finally, a REALIZATION INSTANCE is a regular workflow unpacked from a hyperworkflow; it is a configuration of a hyperworkflow such that all realization variables have been assigned a particular realization value. Hyperworkflows are useful for performing exploration of hyperparameters, ablation studies, variation of input corpora, etc.

For HyperWorkflows, we use a HYPERDAG, the hypergraph formulation of a DAG. shown in Figure 1. In LoonyBin, a HYPEREDGE is an edge originating from a PACKING NODE (displayed as a triangle in Figure 1) , which is used to introduce a realization variable. These packing nodes act like a switch to select one of its input edges so that each edge feeding a packing node can create a new realization variable in the workflow. These realization variables are then propagated through the remainder of the workflow. Where multiple realization variables meet, LoonyBin produces the cross-product of their realization values. A HyperDAG is a packed representation of

multiple workflow DAGs and a realization instance is a particular unpacked instance of a workflow. For instance, in Figure 1 edges *st* and *ch* enter a packing node and then propagate realization values *st* and *ch*. By representing workflows in this way, we avoid rerunning steps having the same experimental conditions.

1.3. Standardized Logging and Organized Directory Structure

While being able to automatically execute and reproduce workflows is good, simply completing the job is not enough. We also want to know where the output files came from and some aggregate facts about them. LoonyBin provides a framework for automatically calculating such information and storing it in a uniform format: tab-delimited key-value pairs form a single record, and each record is newline-delimited, making it easy to process these log files using standard command-line tools or scripts. Finally, the log files for all antecedent steps of the same realization instance are concatenated together so that all information from all steps run under a single experimental condition is collected in one place.

Since the user might want to run further analysis later, it is important to be able to easily find the data itself. To accommodate this, LoonyBin maintains a highly organized directory structure for each workflow. Under a master directory, LoonyBin creates a directory with the name of each vertex in the hyperworkflow with subdirectories for each realization. If steps were run on remote machines, pointers to those machines and the relevant output files are stored on a central machine.

1.4. Designing and Deploying a Workflow

LoonyBin provides a graphical tool, which lists all tools in browsable tree. Tools can simply be dragged and dropped into the workflow as vertices and edges can be drawn by dragging arrows between these vertices.

Once a workflow has been designed, LoonyBin can then compile it into an executable shell script. Thus, the only requirement on the machine that executes the workflow is Bash. Before any tools are ever executed, the generated script checks that all input files and all directories containing required tools exist. Because LoonyBin handles all filenames other than the initial inputs, this eliminates the common issue of pipelines crashing due to typos in file and directory names. The generated script will log into remote machines, copying files and executing processes as necessary.

2. A Machine Translation Toolpack

While LoonyBin provides a mechanism for combining tools into workflows, it does not in itself enable the use of tools. For this, we need tool descriptors, which give LoonyBin 1) what inputs, outputs, and parameters a tool requires 2) analyzers that extract aggregate information from output files and perform sanity checks and 3) documentation on the tool that is shown to the user in the graphic interface. The primary

purpose of the MT Toolpack is to provide these descriptors, their analyzers, and common workflows that put the tools together.

2.1. Installation and Configuration

First, we will set up the DESIGN MACHINE where the visual workflow designer will be used to compile workflows into scripts (e.g. a personal laptop). The only dependency on this machine is Java since the Python tool descriptors are executed via Jython. On this machine, download the latest version of LoonyBin and the MT toolpack[1] and extract the tarballs in the same location. You should now have a `LoonyBin` directory that contains a `tool-packs` directory.

Next, we will set up the EXECUTION MACHINES where the compiled workflow script will be run (e.g. head nodes of various clusters). There, download the MT toolpack and extract the tarball, but also execute the installer script `install-dependencies.py`. This will install *only* the tool binaries, not their dependencies. Other dependencies that must already be installed on the machine include: Python (for the installer), Perl (various), Ruby (Multi-Metric Scorer, MEMT), Java (various), Hadoop (SAMT and Chaksi), Boost (MEMT), and Boost Jam (MEMT). The installer will install these binaries in the user-specified directory and also create a PATHS DIRECTORY, which tells LoonyBin where to find the tool binaries on each execution machine. You can prevent a given tool *X* from being installed by using the `--without-X` switch.

LoonyBin can be launched on most platforms by double-clicking the LoonyBin.jar file. Alternatively, it can be invoked with `java -jar LoonyBin.jar`.

2.2. Creating a Workflow

In this section, we describe the creation of an example workflow. This is done on the DESIGN MACHINE, which need not have any network connection to the machines on which the workflow will run. In "editing" mouse mode, select the "manual filesystem" tool from the panel on the left and then click in the center window to create a vertex in the workflow. Use the panel on the right to give the vertex the name `100-files` (the number in the name is just to help us remember what order the steps were run when looking at the names of vertex subdirectories on the file system) and set the `fileNames` parameter to `example1.txt`. Next, add the Head tool from the left toolbox into the workflow and name it `200-take-head`. Create an edge between the vertices by dragging and, in the Add Edge Dialog that appears, connect `example1.txt` to `corpusIn`.

While we could generate a working script from the workflow created so far, we will continue on and create a HyperWorkflow that demonstrates how to "experiment" with the effect head on 2 different files.

Right-click on the edge from `100-files` to `200-take-head` and select `remove vertex`. Next, add another manual filesystem vertex just as above except with the file-

[1] LoonyBin and the MT Toolpack are available at http://www.cs.cmu.edu/~jhclark/loonybin/

names as `example2.txt` and call it `110-different-files`. Create an OR vertex using the OR tool and give the vertex a unique name. Create a hyperedge from `100-files` to the OR vertex by dragging and, in the Add Edge Dialog that appears, connect `example1.txt` to `OR` and press OK. Similarly, connect `110-different-files` to the OR vertex, and in the dialog connect `example2.txt` to `example1.txt` to indicate that these 2 files will be fulfilling the same role in subsequent steps. Now, in "selecting" mouse mode, click on each of the hyperedges and, using the right panel, name them `one` and `two`, respectively. Finally, draw an edge between the OR vertex and `200-take-head` and connect `example1.txt` as the input of `corpusIn`. You will notice that all of the realization names now appear under the new tool vertex. The tool will be run once for each realization using the inputs from each realization edge.

If you wish multiple tools to feed into the same realization variable, you can give the same name to multiple hyperedges feeding into a single packing vertex. Much like each realization instance had different input files above, you can conduct parameter sweeps using multiple Parameter Boxes from the tool tree on the left; each of the parameter boxes can specify a different set of parameter values to be passed to a tool.

2.3. Generating and Running Workflow Script

LoonyBin allows you to design your pipeline on one machine (the DESIGN MACHINE) and then execute the generated bash script on another machine such as a server – hereafter the HOME MACHINE. The home machine will use passwordless SSH to contact any other remote EXECUTION MACHINES (see Section 2.1).

The "Generate bash script" dialog will ask you for this path of the LoonyBin scripts on the home machine. Also, you need to tell LoonyBin a base directory on the home machine where log data and pointers to output data generated during workflow execution will be placed (see Section 1.3). You should also specify the path and name of the bash script that will be generated. We recommend a `.work` extension. Finally, you can give LoonyBin a space-separated list of email addresses to notify when the pipeline either fails or succeeds. Now just copy the bash generated bash script to the home machine you specified and execute it by passing the `-run` flag. All required input files for each step will automatically be transferred to the proper machine before the tool is executed.

3. Included Tools

We now turn to describing the tools that are included in this MT Toolpack. Since LoonyBin provides documentation within the visual workflow designer for each parameter and file of each tool, we will not focus on the low-level details of the tools here. Instead, we discuss the high-level models they implement and what design decisions were made to incorporate each tool into LoonyBin. In general, the style of LoonyBin is to split tasks into as many LoonyBin tools. This allows easy embedding of novel tools,

resumption on failure, analysis of intermediate results, and sharing partial results in a dynamic programming fashion when later models are run with different parameters.

3.1. MGIZA and Chaksi

MGIZA is a multi-threaded word alignment tool based on GIZA++ (Och and Ney, 2003) that utilizes multiple threads to speed up the time-consuming word alignment process. It also supports forced alignment (the process of aligning an unseen test set given trained models) and incremental training with existing models. It can be distributed over a cluster via its integration with Chaksi, a Hadoop MapReduce training framework for phrase-based machine translation. In addition to word alignment, Chaski supports training of Moses-compatible phrase tables and lexicalized reordering models. In our experience, Chaksi has reduced the time to produce a translation model from parallel data from 4 to 5 days to 9-10 hours. For the initial release of LoonyBin we include tools for generating word classes, both Chaksi and MGIZA versions of the most used word alignment models 1/HMM/3/4, and a phrase table builder. Each of these alignment models is exposed as a separate tool to provide the benefits described above in Section 3.

In building LoonyBin MT tools, we aim to encourage best practice. For instance, MGIZA uses the expectation maximization (EM) algorithm to train word alignment models. In every iteration, the sentences are first aligned using the model parameters from previous step, and then the posteriors are collected and re-normalized to generate models for next step. Therefore, the final alignment output is aligned using the model from second-to-last step instead of the final model. Thus, neither concatenating the sets nor force-aligning using the final model is a good comparison for the way the final model was actually aligned. To encourage proper evaluation of word alignments (by using the second-to-last set of EM parameters), we clearly label the output files that should be used for forced alignment in each tool.

3.2. Berkeley Aligner

The Berkeley Aligner provides an implementation for joint or independent training of IBM Model 1, the HMM alignment model, a syntactic variant of HMM, and a novel symmetrization technique called competitive thresholding (DeNero and Klein, 2007). The aligner provides a supervised inverse transduction grammar (ITG) alignment model (Haghighi et al., 2009). While LoonyBin aims to expose subcomponents as much as possible so that it is easier to combine tools in novel ways, the initial release of the MT toolpack contains only 2 tools for the Berkeley aligner corresponding to the supervised and unsupervised models. In the future, we may attempt to expose each direction, model, and symmetrization heuristic employed in the unsupervised model.

3.3. Joshua

Joshua (Li et al., 2009) is an open-source MT toolkit for synchronous context-free grammar models such as Chiang (2005). It includes suffix array extraction of these grammars from an aligned parallel corpus. The toolkit also includes a built-in subsampler for training on large corpora and an implementation of minimum error-rate training. Each step in the training pipeline is exposed as a separate tool in the LoonyBin MT Toolpack.

3.4. Syntax-Augmented Machine Translation (SAMT)

The SAMT model (Zollmann and Venugopal, 2006) is a synchronous context-free grammar based approach to translation that extends the hierarchical phrase based MT model of (Chiang, 2005) to learn grammars with multiple nonterminals. Grammar rules are extracted from a training sentence pair based on a lattice of its contained eligible phrase pairs and a phrase-structure parse tree of the target sentence, yielding rules such as

NP+SBAR $\rightarrow$ NP , die meine NN zuletzt VBD | NP who last VBD my NN

for a German-to-English translation task, expressing the reordering of the verb triggered by a relative clause. The current release of SAMT uses the open-source Hadoop MapReduce framework to distribute its expensive computations (Venugopal and Zollmann, 2009). Each step in the SAMT training and evaluation pipeline has been wrapped as a separate tool in the LoonyBin MT Toolpack.

3.5. Moses

We replace the `train-phrase-model.perl` from Moses (Koehn et al., 2007) with tools that encapsulates each step such as "build lexical translation table," "construct lexicalized reordering model," and "Run Minimum Error Rate Training" rather than wrapping the entire pipeline. Steps that use GIZA++ are not included in the MT Toolpack since with the release of MGIZA++ and Chaksi, there is little motivation to use GIZA++. For the initial release of the MT toolpack, we do not support factored models.

3.6. Common Evaluation Metrics

We provide a tool that runs some of the most common translation metrics in parallel while transparently handling formatting issues: BLEU (Papineni et al., 2001) as implemented by `mteval-13a.pl` (Peterson et al., 2009), NIST (Doddington, 2002), TER 0.7.25 (Snover et al., 2006), Meteor 1.0 (Banerjee and Lavie, 2005), unigram precision and recall, and length ratio. It accepts a simple input format: flat files with one line per segment, or consecutive lines for multiple references. Aside from translation metrics,

we also include alignment error rate (AER) (Och and Ney, 2003), despite its imperfect correlation with translation quality. In addition to providing the files generated by each metric as output, the LoonyBin tool descriptor places all of these scores in the LoonyBin log giving the benefit of standard formatting.

3.7. Multi-Engine Machine Translation (MEMT)

Multi-engine machine translation (Heafield et al., 2009) combines one-best outputs from different translation systems. Translations are aligned using METEOR (Banerjee and Lavie, 2005) and navigated using these alignments. System-specific weights are learned via tuning with MERT; a separate tuning set works best. Typical gains range from one to five BLEU points above the best system, depending on system diversity and score distribution. MEMT is presented as three tools in LoonyBin: The Meteor aligner, MEMT Tuning, and MEMT Decoding.

3.8. Additional NLP Tools

Since modern MT systems often depend on more basic NLP tools, we have also included a few of these tools in the MT Toolpack. For creating language models, we include SRILM and for creating parse trees, we include the Stanford English parser.

4. Recommendations During Tool Development

LoonyBin aims to make it easy to reproduce results. Well-behaved tool descriptors should write the software version to the log files so that the user knows not only what files were used as input and what tools processed that data, but also what version of the tools were used.

However, research often involves iteratively coding and experimentation. For this, we recommend creating a custom tool descriptor that checks out your branch of a source code management system (e.g. subversion), logs the revision number, compiles the code, and then runs the tool. By doing this, researchers can ensure that results are reproducible[2]. Step-by-step instructions on how to create tool descriptors are included as part of LoonyBin's documentation, but are beyond the scope of this paper.

5. Conclusion

We have presented an open-source Machine Translation Toolpack for LoonyBin. We hope that by releasing this tool pack more research effort may be placed on modeling rather engineering, automation, and logging. Further, we hope that this toolpack encourages future research to include the multiple baseline systems and enables more systematic comparisons between them.

[2] As a side benefit, this encourages the best practice of "commit early, commit often"

Bibliography

Banerjee, S. and A. Lavie. METEOR: an automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, 2005.

Chiang, David. A hierarchical phrase-based model for statistical machine translation. In *Proceedings of the 43rd Annual Meeting on Association for Computational Linguistics*, 2005.

Clark, Jonathan H. and Alon Lavie. Loonybin: Keeping language technologists sane through automated management of experimental (hyper)workflows. In *Forthcoming*, 2010.

DeNero, J. and D. Klein. Tailoring word alignments to syntactic machine translation. In *Association for Computational Linguistics (ACL)*, volume 45, page 17, 2007.

Doddington, G. Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. In *Proceedings of the second international conference on Human Language Technology Research*, page 145. Morgan Kaufmann Publishers Inc., 2002.

Haghighi, A., J. Blitzer, J. DeNero, and D. Klein. Better word alignments with supervised ITG models. In *Meeting of the Association for Computational Linguistics*, 2009.

Heafield, Kenneth, Greg Hanneman, and Alon Lavie. Machine translation system combination with flexible word ordering. In *Proceedings of the Fourth Workshop on Statistical Machine Translation*, pages 56–60, Athens, Greece, March 2009. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W09/W09-0x08`.

Koehn, Philipp, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *Association for Computational Linguistics (ACL)*, 2007.

Li, Zhifei, Chris Callison-Burch, Chris Dyer, Juri Ganitkevitch, Sanjeev Khudanpur, Lane Schwartz, Wren Thornton, Jonathan Weese, and Omar Zaidan. Joshua: An open source toolkit for parsing-based machine translation. In *Workshop on Statistical Machine Translation (WMT09)*, 2009.

Och, Franz Josef and Hermann Ney. A systematic comparison of various statistical alignment models. In *Computational Linguistics*, 2003.

Papineni, K., S. Roukos, T. Ward, and W. J Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proc. of ACL*, 2001.

Peterson, Kay, Mark Przybocki, and Sébastien Bronsart. NIST 2009 open machine translation evaluation (MT09) official release of results, 2009. http://www.itl.nist.gov/iad/mig/tests/mt/2009/.

Snover, M., B. Dorr, R. Schwartz, L. Micciulla, and J. Makhoul. A study of translation edit rate with targeted human annotation. In *Proc. of AMTA*, page 223–231, 2006.

Venugopal, Ashish and Andreas Zollmann. Grammar based statistical MT on hadoop. *The Prague Bulletin of Mathematical Linguistics*, 91, 2009.

Zollmann, Andreas and Ashish Venugopal. Syntax augmented machine translation via chart parsing. In *Workshop on Machine Translation (WMT) at ACL*, 2006.