PBML

logo

The Prague Bulletin of Mathematical Linguistics

NUMBER ??? OCTOBER 2012 1–10

An Experimental Management System

Philipp Koehn

University of Edinburgh

Abstract

We describe Experiment.perl, an experimental management system, that allows the execution of the entire training and testing pipeline of a machine translation experiment with one configuration files. When carrying out multiple experimental runs with changed settings, Experiment.perl automatically detects which steps need to be re-run and which can be re-used.

1. Introduction

Running a machine translation experiment involves many steps: preparing training data, building language and translation models, tuning, testing, scoring and analysis of the results. For most of these steps, a different tool needs to be invoked, so this easily becomes very cumbersome. The Experiment Management System (EMS), or Experiment.perl, for lack of a better name, makes it much easier to run experiments. It ships with the Moses machine translation toolkit (**?**).

A typical example is given in Figure 1. The graph was automatically generated by Experiment.perl. All that needed to be done was to specify one single configuration file that points to data files and settings for the experiment. In the graph, each step is a small box. For each step, Experiment.perl builds a script file that gets either submitted to a compute cluster or executed on the same machine. Note that some steps are quite involved, for instance tuning: On a cluster, the tuning script runs on the head node a submits jobs to the queue itself.

Experiment.perl makes it easy for multiple experimental runs with different settings]. It automatically detects which steps do not have to be executed again. Experiment.perl plays the same role as LoonyBin (**?**), but there are significant differences in its usage. Experiment.perl uses a textual configuration file to set up an experiment and

Corresponding author: pkoehn@inf.ed.ac.uk
Cite as: Philipp Koehn. An Experimental Management System. The Prague Bulletin of Mathematical Linguistics No. ???, 2012, pp. 1–10.

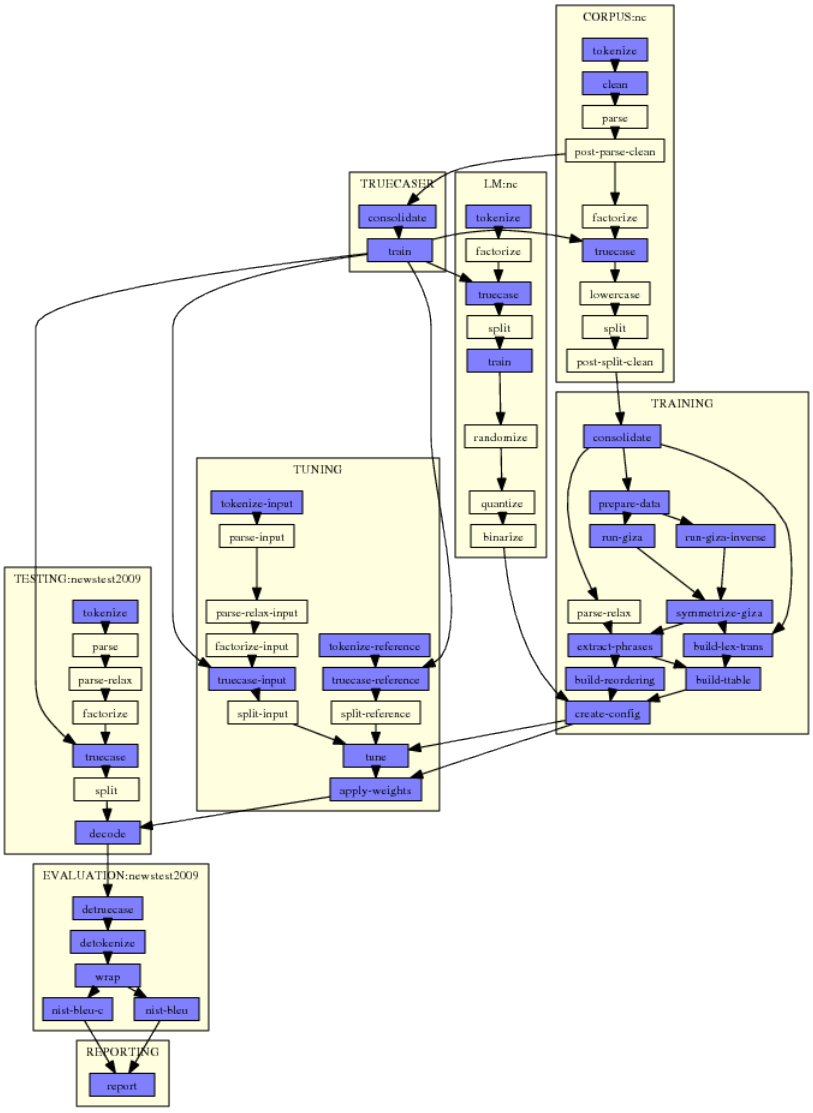

Figure 1. **Workflow generated by Experiment.perl:** *Steps (such as* `run-giza` *are grouped into modules (such as* `TRAINING`*). Dependencies are indicated by arrows. Skipped steps have a pale background.*

a meta configuration file to define all possible workflows, LoonyBin offers a graphical user interface that lets the user connect steps.

2. Design

Experiment.perl breaks up training, tuning, and evaluating of a statistical machine translation system into a number of steps, which are then scheduled to run in parallel or sequence depending on their inter-dependencies and available resources. The possible steps are defined in the file `experiment.meta`. An experiment is defined by a configuration file.

2.1. Experiment.Meta

The actual steps, their dependencies and other salient information is to be found in the file `experiment.meta`. Think of `experiment.meta` as a "template" file. Steps are grouped into modules, which are:

- `CORPUS`: preparing a parallel corpus
- `INPUT-FACTOR` and `OUTPUT-FACTOR`: commands to create factors
- `TRAINING`: training a translation model
- `LM`: training a language model
- `INTERPOLATED-LM`: interpolate language models
- `SPLITTER`: training a word splitting model
- `RECASING`: training a recaser
- `TRUECASING`: training a truecaser
- `TUNING`: running minumum error rate training to set component weights
- `TESTING`: translating and scoring a test set
- `REPORTING`: compile all scores in one file

To give an example of a step definition in `experiment.meta`, here the parts of the definition for `LM:get-corpus` and `LM:tokenize`:

```
get-corpus
       in: get-corpus-script
       out: raw-corpus
       [...]

tokenize
       in: raw-corpus
       out: tokenized-corpus
       [...]
```

Each step takes some input (`in`) and provides some output (`out`). This also establishes the dependencies between the steps. The step `tokenize` requires the input `raw-corpus`. This is provided by the step `get-corpus`.

The file `experiment.meta` provides a generic template for steps and their interaction. For an actual experiment, a configuration file determines which steps need to be run. This configuration file is specified when invoking `experiment.perl`. It may contain for instance the following:

```
[LM:europarl]

### raw corpus file
#
raw-corpus = $europarl-v3/training/europarl-v3.en
```

Here, the corpus to be used for language modeling is named `europarl` and it is provided in raw text format in the location `$europarl-v3/training/europarl-v3.en` (the variable `$europarl-v3` is defined elsewhere in the config file). The effect of this specification in the config file is that the step `get-corpus` does not need to be run, since its output is given as a file. The workflow starts with the step `tokenize`.

The entire definition of an experiment follows this logic, which is very similar to the principles of a Unix Makefile. The ultimate purpose of an experiment is to generate a result file at the end. If this is not given, then scoring scripts need to be called. Scoring scripts require the output of the decoder on the test sets. The decoder requires a tuned model. Tuning requires a trained model. Trained models require language models and model files, and so on. The workflow, as show in Figure 1, is generate bottom up, following the input/output dependencies of steps.

2.2. Elements of Step Definitions

Several parameters for step definitions are used in `experiment.meta`:

- `in` and `out`: Established dependencies between steps; input may also be provided by files specified in the configuration.
- `default-name`: Name of the file in which the output of the step will be stored.
- `template`: Template for the command that is placed in the execution script for the step.
- `template-if`: Potential command for the execution script. Only used, if a specified setting exists.
- `error`: Experiment.perl detects if a step failed by scanning `STDERR` for key words such as killed, error, died, not found, and so on. Additional key words and phrase are provided with this parameter.
- `not-error`: Declares default error key words as not indicating failures.
- `pass-unless`: Only if the given setting is used, this step is executed, otherwise the step is passed (illustrated by a yellow box in the graph).
- `ignore-unless`: If the given setting is used, this step is not executed. This overrides requirements of downstream steps.

- `rerun-on-change`: If similar experiment are runs, the output of steps may be used, if input and settings are the same. This specifies settings whose change disallows a re-use in different run.
- `parallelizable`: When running on a cluster or a multi-core machine, this step may be parallelized (only if `generic-parallelizer` is set in the config file).
- `qsub-script`: If running on a cluster, this step is run on the head node, and not submitted to the queue (because it submits jobs itself).

To complete our example, the full definition of the step `LM:tokenize` is below.

```
tokenize
        in: raw-corpus
        out: tokenized-corpus
        default-name: lm/tok
        pass-unless: output-tokenizer
        template: $output-tokenizer < IN > OUT
        parallelizable: yes
```

The step takes `raw-corpus` and produces `tokenized-corpus`. It is parallizable with the generic parallelizer. The output is stored in the file according to the definition `corpus/tok`. Note that the actual file name also contains the corpus name, and the run number. In our example the tokenized corpus is stored in a file named `lm/europarl.tok.1`. The step is only executed, if `output-tokenizer` is specified. The template indicate how the command lines in the execution script for the steps are formed.

2.3. Multiple Corpora, One Translation Model

We may use multiple parallel corpora for training a translation model or multiple monolingual corpora for training a language model (or use multiple language models). Each of these have their own instances of the `CORPUS` and `LM` module. There may be also multiple test sets in `TESTING`). However, there is only one translation model and hence only one instance of the `TRAINING` module. The definitions in `experiment.meta` reflects the different nature of these modules. For instance `CORPUS` is flagged as `multiple`, while `TRAINING` is flagged as `single`.

When defining settings for the different modules, the singular module `TRAINING` has only one section, while this one general section and specific `LM` sections for each training corpus. In the specific section, the corpus is named, e.g. `LM:europarl`. When looking up the parameter settings for a step, first the set-specific section (`LM:europarl`) is consulted. If there is no definition, then the module definition (`LM`) and finally the general definition (in section `GENERAL`) is consulted. In other words, local settings override global settings.

2.4. Configuration File

A configuration file for an experimental run consists of a collection of settings, one per line with empty lines and comment lines for better readability, organized in sections for each of the modules.

The start of each section is indicated by the section name in square brackets (`[TRAINING]` or `[CORPUS:europarl]`). If the word `IGNORE` is appended to a section definition, then the entire section is ignored.

The syntax of setting definition is `setting = value` (note: spaces around the equal sign). If the value contains spaces, it must be placed into quotes (`setting = "the value"`), except when a vector of values is implied (only used when defining list of factors: `output-factor = word pos`. Comments are indicated by a hash (`#`).

Settings can be used as variables to define other settings:

```
working-dir = /home/pkoehn/experiment
wmt10-data = $working-dir/data
```

Variable names may be placed in curly brackets for clearer separation:

```
wmt10-data = ${working-dir}/data
```

Such variable references may also reach other modules:

```
[RECASING]
tokenized = $LM:europarl:tokenized-corpus
```

Finally, reference can be made to settings that are not defined in the configuration file, but are the product of the defined sequence of steps. Say, in the above example, `tokenized-corpus` is not defined in the section `LM:europarl`, but instead `raw-corpus`. Then, the tokenized corpus is produced by the normal processing pipeline. Such an intermediate file can be used elsewhere:

```
[RECASING]
tokenized = [LM:europarl:tokenized-corpus]
```

Some error checking is done on the validity of the values in the configuration file before an experimental run is executed. All values that seem to be file paths trigger the existence check for such files. A file with the prefix of the value must exist.

2.5. Step Files

Let us follow our example of the tokenization step in the language model module in more detail. Recall that the `LM:europarl` section has a specification of `raw-corpus` to `$europarl-v3/training/europarl-v3.en`. Since only the raw corpus, but not a tokenized corpus is specified, Experiment.perl concludes that it needs to run the tokenization step.

The directory steps contains the script that executes each step, its STDERR and STDOUT output, and meta information:

```
steps/1/LM_europarl_tokenize.1
steps/1/LM_europarl_tokenize.1.DONE
steps/1/LM_europarl_tokenize.1.INFO
steps/1/LM_europarl_tokenize.1.STDERR
steps/1/LM_europarl_tokenize.1.STDERR.digest
steps/1/LM_europarl_tokenize.1.STDOUT
```

The file steps/1/LM_europarl_tokenize.1 is the shell script that is run to execute the step. The file with the extension DONE is created when the step is finished - this communicates to the scheduler that subsequent steps can be executed. The file with the extension INFO contains meta information - essential the settings and dependencies of the step. This file is checked to detect if a step can be re-used in subsequent experimental runs.

In case that the step crashed, we expect some indication of a fault in STDERR (for instance the words core dumped or killed). This file is checked to see if the step was executed successfully, so subsequent steps can be scheduled or the step can be re-used in new experiments. Since the STDERR file may be very large (some steps create Megabytes of such output), a digested version is created in STDERR.digest. If the step was successful, it is empty. Otherwise it contains the error pattern that triggered the failure detection.

2.6. Re-Use of Steps

Let us now take a closer look at re-use. If we run the experiment again but change a settings, say, the order of the language model, then there is no need to re-run the tokenization, but only language model training.

Here is the definition of the language model training step in experiment.meta:

```
train
  in: split-corpus
  out: lm
  default-name: lm/lm
  ignore-if: rlm-training
  rerun-on-change: lm-training order settings
  template: $lm-training -order $order $settings -text IN -lm OUT
  error: cannot execute binary file
```

The mention of order in the list behind rerun-on-change informs experiment.perl that this step does need to be re-run, if the order of the language model changes. Since none of the settings in the chain of steps leading up to the training have been changed, those can be re-used.

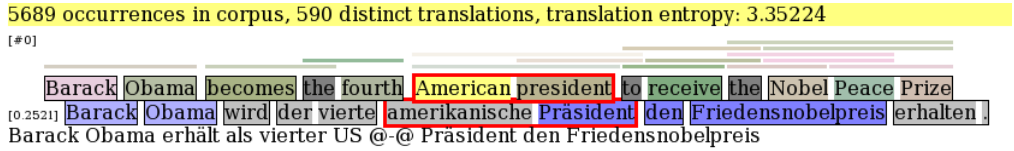

Figure 2. Comparing outputs from two experimental runs

If the language model order is changed and the experiment.perl is run again in the same working directory, you will see the following files in the directory `lm`:

```
% ls -tr lm/*
lm/europarl.tok.1
lm/europarl.truecased.1
lm/europarl.lm.1
lm/europarl.lm.2
```

Note that a new language model was trained for this second run (`lm/europarl.lm.2`), but no new tokenized and truecased corpus files. These were re-used from run 1.

Steps are re-used from previous runs, unless settings listed under `rerun-on-change` are changed, one of its specified input files (if any) are changed, and if one of its previous steps are re-run. Note that if a filename is not changed, but its time stamp differs, this triggers re-running a step. This ensures that only a minimum number of steps are run to produce the exact same outcome as if all steps are run.

2.7. Web Interface and Analysis

Experiment.perl also offers a web interface to the experimental runs for easy access and comparison of experimental results. The web interface gives a listing of experiments and runs for each experiments, with a display of automatic metric scores, and links to configuration files and outputs.

You can include additional analysis for an experimental run in the web interface by specifying the setting `analysis` in its configuration file. This adds reports n-gram precision and recall statistics and color-coded n-gram correctness markup for the output sentences to the web interface. See Figure 2 for an example.

The output is color-highlighted according to n-gram matches with the reference translation. The following colors are used. The darker the color of an output word, the higher n-gram match to the reference translation it is part of.

Additional reports are available when adding the settings `analyze-coverage` and `report-segmentation`. The setting `analyze-coverage` include a coverage analysis: which words and phrases in the input occur in the training data or the translation table? This is reported in color coding and in a yellow report box when moving the

mouse of the word or the phrase. Also, summary statistics for how many words occur how often are given, and a report on unknown or rare words is generated.

The setting `report-segmentation` creates summary statistics about what kind of phrase mappings are used (one word to one word, 1-2, 2-2, etc.), as well as markup of the sentence pair with the phrase segmentation. The phrase segmentation is indicated with black boxes around the words, and the alignment is shown when moving the mouse on the phrases.

3. Usage

3.1. Quick Start

Experiment.perl is extremely simple to use:

- Find `experiment.perl` in `scripts/ems`
- Get a sample configuration file from someplace (for instance `scripts/ems/example/config.toy`).
- Set up a working directory for your experiments for this task (`mkdir` does it).
- Edit the following path settings in `config.toy`
  - `working-dir`
  - `data-dir`
  - `moses-script-dir`
  - `moses-src-dir`
  - `srilm-dir`
  - `decoder`
- Run `experiment.perl -config config.toy` from your working directory.
- Marvel at the graphical plan of action.
- Run `experiment.perl -config config.toy -exec`.
- Check the results of your experiment (in `evaluation/report.1`)

3.2. More Examples

The `example` directory contains some additional examples. These require the training and tuning data released for the Shared Translation Task for WMT 2010.

The examples using these corpora are

- a basic phrase based model
- a factored phrase based model
- a hierarchical phrase based model
- a target syntax model

The factored model using all the available corpora is identical to the Edinburgh submission (**?**) to the WMT 2010 shared task for English-Spanish, Spanish-English, and English-German language pairs. The French language pairs also used the 10^9 corpus, the Czech language pairs did not use the POS language model, and German-English used additional pre-processing steps.

4. Outlook

We have been using Experiment.perl for years and are satisfied with its core functionalities. In future work, we would like support job scheduling on Hadoop clusters and extend the analysis facility.

Acknowledgments

This work was supported in part by the EuroMatrixPlus project funded by the European Commission (7th Framework Programme) and in part under the GALE program of the Defense Advanced Research Projects Agency, Contract No. HR0011-06-C-0022.

Address for correspondence:
Philipp Koehn
pkoehn@inf.ed.ac.uk
School of Informatics
University of Edinburgh
Scotland, United Kingdom