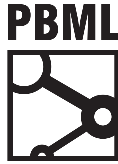


The Prague Bulletin of Mathematical Linguistics
NUMBER 98 OCTOBER 2012 51–61

pycdec: A Python Interface to cdec

Victor Chahuneau, Noah A. Smith, Chris Dyer

Language Technologies Institute, Carnegie Mellon University

Abstract

This paper describes `pycdec`, a Python module for the `cdec` decoder. It enables Python code to use `cdec`'s fast C++ implementation of core finite-state and context-free inference algorithms for decoding and alignment. The high-level interface allows developers to build integrated MT applications that take advantage of the rich Python ecosystem without sacrificing computational performance. We give examples of how to interact directly with the main `cdec` data structures (lattices, hypergraphs, sparse feature vectors), evaluate translation quality, and use the suffix-array grammar extraction code. This permits rapid prototyping of new algorithms for training, data visualization, and utilizing MT and related structured prediction tasks.

1. Introduction

Machine translation decoders are complex pieces of software. They must provide efficient search and inference algorithms, represent large translation grammars (e.g., phrase tables), and support scoring of hypotheses with a variety of feature functions. Typically, they also contain functionality for parameter learning and translation quality evaluation. Despite this sophistication, machine translation can be formalized quite well using familiar, well-defined mathematical objects (e.g., lattices, vectors, hypergraphs, weighted finite-state transducers) and in terms of just a few algorithms (e.g., FST/CFG intersection, shortest path search, etc.).

Although this convenient and precise mathematical language exists (and is, of course, used in the academic literature), the programmatic interfaces to real translation systems are much more complicated. On one hand, the low-level implementation in the decoder's native language (usually C++ or Java) is highly optimized, making the mapping between the mathematical primitives discussed in papers and the actual code difficult to perceive. On the other hand, the high-level command-line interface

© 2012 PBML. All rights reserved.
Corresponding author: `vchahune@cs.cmu.edu`
Cite as: Victor Chahuneau, Noah A. Smith, Chris Dyer. pycdec: A Python Interface to cdec. The Prague Bulletin of Mathematical Linguistics No. 98, 2012, pp. 51–61. doi: 10.2478/v10108-012-0008-7.

that decoders expose is not suitably expressive for anything but the most coarse automation. As a result, when new researchers and engineers master the theory of MT, they must still invest a great deal of work in learning a real software system before they can really innovate. This paper describes a new Python interface for the `cdec` decoder designed to narrow the gap between theory and practice.

`cdec` is a good candidate for this task because it has been designed with modularity in mind from the beginning (Dyer et al., 2010). We choose Python as the language to expose the API for its large user base and rich extension ecosystem, and also because it is an interpreted language supporting both object-oriented and functional programming. The goals for this project include:

- exposing the decoder functionality as a library with a natural, easy-to-understand interface;
- providing access to the decoder's data structures, including translation hypergraphs, input lattices, hypothesis feature vectors, etc.;
- allowing direct integration of external Python libraries such as *NLTK* (Bird et al., 2009) and *scikit-learn* (Pedregosa et al., 2011) into machine translation systems; and
- encouraging creative use of machine translation technology by programmers who do not need to learn the details of open-source machine translation systems.

The `pycdec` interface is implemented using Cython (Behnel et al., 2011) and included as part of the open-source `cdec` distribution.[1] In the following, we give an introduction to its main functionality and then describe a few applications of the new interface.

2. Related Work

Experiment management tools (Koehn, 2010; Clark et al., 2010) abstract the internals of the decoder from the user to provide a uniform interface to the main training steps of the system. While these facilitate the coordination of large experimental setups, they must be configured using either a domain-specific language or a graphical interface that the user has to learn to manipulate the system. We go in the opposite direction and directly expose the decoder to the user in a modern and familiar language, Python.

Recent work has also explored the use of visualization tools for machine translation. Weese and Callison-Burch (2010) describe extensions to the Joshua decoder to populate a graphical interface used to display derivation trees and hypergraphs. We obtain similar functionality with just a couple of lines of `pycdec` in conjunction with existing visualization tools (§ 4.1). Since our visualizations are computed with simple Python scripts, developers have far more flexibility to innovate.

Finally, the popularity of web translation services such as Google Translate has motivated the development of web interfaces for open-source translation tools (Fed-

[1] `http://cdec-decoder.org`

ermann and Eisele, 2010). We demonstrate how such tools can be rapidly developed using common networking and communication libraries (§ 4.2).

3. Library Description

The API of pycdec exposes the main data structures and algorithms necessary for machine translation and similar structured prediction problems. When it makes sense to do so, we retain the structure of the C++ interface, but otherwise follow the Python conventions.

3.1. Basic Translation and Inference API

The translation interface is provided by the Decoder class. The constructor takes arguments specifying the configuration of the decoder. Feature weights used by the decoder can be assigned and modified at any time (for example, in an online training algorithm).

Once the decoder is instantiated, it can translate sentences, optionally using a sentence-specific grammar passed as a string argument. The result returned is a translation hypergraph encoding the search space explored by the decoder.

The Hypergraph object is central to this system, and therefore it supports several types of operations:

- extraction of the Viterbi translation (viterbi), source and target trees (viterbi_trees) and of the corresponding feature vector (viterbi_features);
- extraction of k-best translations (kbest), source and target trees (kbest_trees) and of the corresponding feature vectors (kbest_features);
- operations that modify the hypergraph, including:
  - rescoring with new weights (reweight),
  - inside-outside pruning (prune),
  - intersection with a reference sentence or lattice (intersect); and
- iteration over the edges and nodes that form the hypergraph.

As an example, here is how to use a hierarchical phrase-based decoder to translate a sentence with a grammar read from a file:

```
import cdec
# Create and configure a decoder object
decoder = cdec.Decoder(formalism='scfg',
        feature_function=['WordPenalty', 'KLanguageModel lm.klm'],
        add_pass_through_rules=True)
# Set weights for the language model features
decoder.weights['LanguageModel_OOV'] = -1
decoder.weights['LanguageModel'] = 0.1
# Read a SCFG from a file
grammar = open('grammar.scfg').read()
```

```
# Translate the sentence; returns a translation hypergraph
hg = decoder.translate('traduttore , traditore .', grammar=grammar)
# Extract the best hypothesis from the hypergraph
print(hg.viterbi())
```

Other formalisms such as phrase-based translation can be accessed in a similar way by setting the appropriate configuration parameters for the decoder.

3.2. Grammar Extraction API

To minimize memory usage and code complexity, cdec uses *per-sentence grammars* (i.e., grammars containing just the rules that can match the words in a single test sentence). While these grammars can be constructed from arbitrary tools, pycdec includes the suffix array grammar extractor of Lopez (2007), which uses an efficient compiled representation of a parallel corpus and word alignment to construct translation grammars on demand for new test sentences. The Python module makes this online grammar extraction procedure particularly simple.

After the training corpus has been compiled into a suffix array representation using the tools distributed with cdec, the resulting configuration can be used to call the grammar extractor for any arbitrary input:

```
extractor = cdec.sa.GrammarExtractor('extractor_config.py')
decoder = cdec.Decoder(formalism='scfg')
sentence = 'traduttore , traditore .'
decoder.translate(sentence, grammar=extractor.grammar(sentence))
```

The extraction algorithm is implemented in Cython and is suitable for online extraction of grammars from very large corpora (Lopez, 2008).

3.3. Translation Quality Evaluation

cdec includes implementations of basic evaluation metrics (BLEU and TER), exposed in Python via the cdec.score module. For a given (reference, hypothesis) pair, sufficient statistics vectors (SufficientStats) can be computed. These vectors are then added together for all sentences in the corpus and the final result is finally converted into a real-valued score.

Writing a script which computes the BLEU score for a set of hypotheses and references is thus straightforward:

```
import cdec.score
with open('hyp.txt') as hyp, open('ref.txt') as ref:
    stats = sum(cdec.score.BLEU(r).evaluate(h) for h, r in zip(hyp, ref))
    print('BLEU = {0:.1f}'.format(stats.score * 100))
```

Multiple references can be used by supplying a list of strings instead of a single string:

```
cdec.score.BLEU([r1, r2])
```

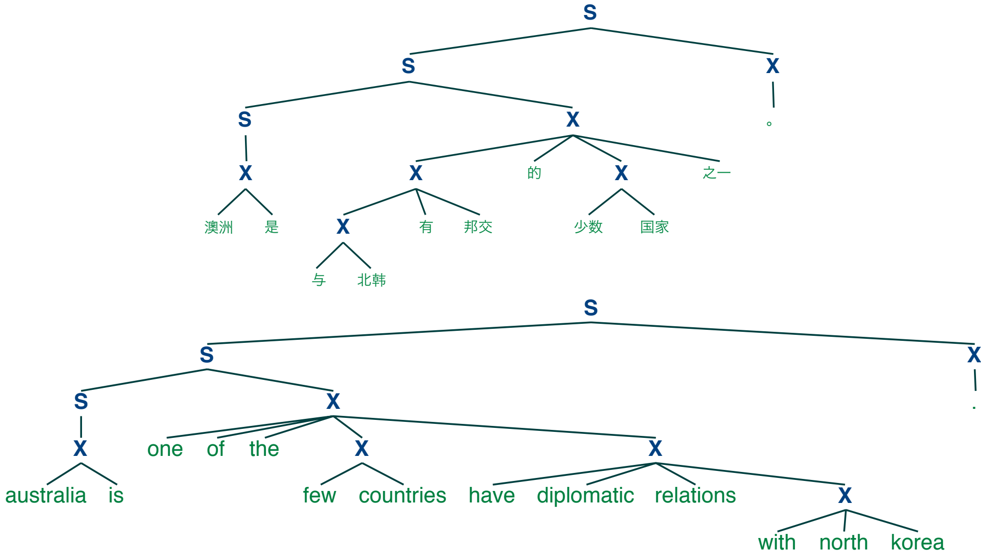


Figure 1. Source (Chinese) and target (English) parse trees, drawn using NLTK

When implementing training algorithms using `pycdec` (§ 4.3), it is often necessary to manipulate k-best lists of scored hypotheses. For every metric, sentence scorers are able produce such sets of hypotheses (`CandidateSet`). For each `Candidate` in the list, its sentence-level metric score (`score`), feature vector (`fmap`) and output string (`words`) can be obtained.

4. Applications

In this section, we provide several examples using the `pycdec` module to solve visualization, parameter estimation, and grammar extraction problems.

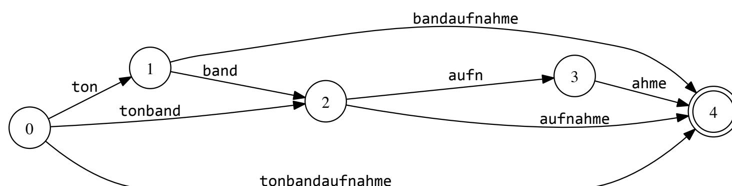
4.1. Visualizing the Result of Decoding

We can make use of the functionality of *NLTK* to visualize derivation trees that result from the decoding of a sentence under a synchronous grammar. Fig. 1 shows an example for a Chinese/English hierarchical phrase-based system. The corresponding Python code is:

```
hg = decoder.translate(sentence)
f_tree, e_tree = hg.viterbi_trees()
nltk.Tree(f_tree).draw() # draw source tree
nltk.Tree(e_tree).draw() # draw target tree
```

Another finite-state formalism supported by `cdec` is compound splitting, in which case the model output takes the form of a lattice (encoded as a hypergraph produced by the `translate` method). Conversion to the Graphviz dot format (Ellson et al., 2003) allows a compact visualization of the output space. Then we can use any of the several Python interfaces to Graphviz to directly render the lattice as shown below:

```
hg = decoder.translate('tonbandaufnahme')
hg.prune(beam_alpha=9.0, csplit_preserve_full_word=True)
pydot.graph_from_dot_data(hg.lattice().todot()).write_svg('lattice.svg')
```



Finally, we introduce a more complex visualization which makes use of the direct access to the hypergraph (Fig. 2). For the same sentence as our first example, we represent the synchronous parse chart as a table, with each cell containing all the possible non-terminals for the corresponding span. Then we color the background of the cell according to the following value:

$$\log \sum_{\text{node} \in \text{nodes}} \max_{\text{edge} \to \text{node}} p(\text{edge})$$

This gives an indication of how much uncertainty is present at each level of the parse. We believe that this is an efficient method to compactly visualize the enormous output space produced by the decoder: the hypergraph contains 244,232 edges and 77 nodes encoding a total of 3.8×10^{28} paths!

We conclude by noting that, as opposed to specialized visualization tools (e.g., Weese and Callison-Burch, 2010), `pycdec` allows the programmer to use any algorithm and output format to explore the various decoder data structures. We suggest in particular the use of the IPython notebook (Pérez and Granger, 2007) to produce HTML or SVG graphics directly in a web browser, as we did for Fig. 2.

4.2. A Web Translation Interface

Commercial web translation platforms, such as Google Translate, have been very successful in bringing state of the art machine translation systems to internet users. In a research environment, it can also be useful to provide similar web interfaces, for example, for non-technical users to explore the strength and weaknesses of the system.

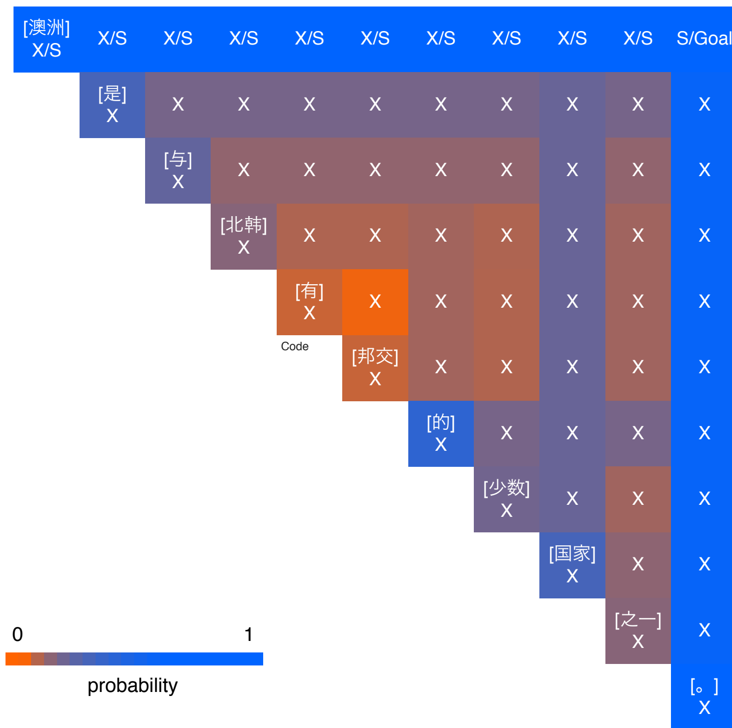


Figure 2. Chart for the synchronous parse of a Chinese sentence

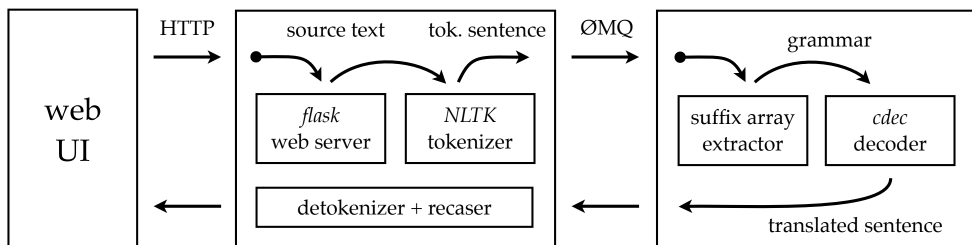


Figure 3. Architecture of the web translation service

Since pycdec provides an access to the decoder directly from Python, it is possible to implement such a service with standard networking libraries to manage communication. Fig. 3 illustrates the messages transmitted between the three layers of the architecture as a piece of text is translated:

- The user interface consists of a HTML page with a JavaScript UI interacting with the web server via asynchronous HTTP requests.
- When the web server – a Python application implemented using the *flask* web framework – receives a translation request, it applies standard pre-processing steps to the input. The text is first segmented into sentences, and each sentence is in turn tokenized. We rely on *NLTK* for this step, at least when the source language is English. Then, each sentence is sent separately through a ZeroMQ[2] socket to the translation server, using the *pyzmq* library.
- The translation server receives individual sentences, for which it extracts grammars on the fly as explained in § 3.2, before calling the decoder to translate the sentence with the extracted grammar. It replies to the web server with the translated sentence.
- The web server then post-processes each translated sentence and recomposes the translated text block before transmitting it back to the web UI.

Even with such a minimal architecture, our system can easily be scaled by multiplying the number of translation servers and relying on ZeroMQ to distribute translation tasks to the multiple decoder instances.

4.3. Parameter Estimation

Another natural use case for pycdec is to facilitate development of new discriminative parameter learning algorithms in Python. Such algorithms (e.g., Chiang et al., 2008; Hopkins and May, 2011; Gimpel and Smith, 2012) use the decoder to compute statistics over the hypergraphs or k-best lists produced by decoding a development set so as to optimize some objective function (like BLEU, or likelihood). In these algorithms, the majority of the computational effort is the decoding step (or a similar inference problem, such as computing posterior probabilities over n-grams), whereas the manipulation of the weight vector is inexpensive. Thus, a natural division of labor is to use Python's mathematical libraries for manipulation of the weight vector and pycdec for inference.

Advantages of writing a new training method with pycdec include the possibility to easily debug code by directly interacting with the decoder data structures through the Python interpreter, and the availability of mature machine learning libraries such as *scikit-learn*.

To illustrate these claims, we implement a recently published training method that is not currently included in cdec. We choose Bazrafshan et al. (2012), a simple exten-

[2] http://www.zeromq.org

```python
decoder = cdec.Decoder(...)

def get_pairs(source, reference):
    hg = decoder.translate(source)
    # 1. Generate a list containing the k best translations
    cs = cdec.score.BLEU(reference).candidate_set()
    cs.add_kbest(hg, K)
    # 2. Use the uniform distribution to sample n random pairs
    # from the set of candidate translations
    pairs = []
    for _ in range(n_samples):
        ci = cs[random.randint(0, len(cs) - 1)]
        cj = cs[random.randint(0, len(cs) - 1)]
        # 3. Keep a pair of candidates if the difference between their score
        # is bigger than a threshold t
        if abs(ci.score - cj.score) < score_threshold: continue
        pairs.append((ci.fmap - cj.fmap, ci.score - cj.score))
    # 4. From the potential pairs kept in the previous step,
    # keep the s pairs that have the highest score
    for x, y in heapq.nlargest(n_pairs, pairs, key=lambda xy: abs(xy[1])):
        # 5. For each pair kept in step 4, make two data points
        yield x, y
        yield -1 * x, -1 * y

# The DictVectorizer converts dictionaries into sparse vectors
vectorizer = sklearn.feature_extraction.DictVectorizer()

for _ in range(n_iterations):
    # Collect training pairs
    X, g = [], []
    for source, reference in zip(sources, references):
        for x, y in get_pairs(source, reference):
            X.append(dict(x))
            g.append(y)
    # Train a linear regression model
    model = sklearn.linear_model.LinearRegression()
    X = vectorizer.fit_transform(X)
    model.fit(X, g)
    # Update weights with the learned model
    for fname, fval in zip(vectorizer.feature_names_, model.coef_):
        decoder.weights[fname] = (alpha * fval +
                (1 - alpha) * decoder.weights[fname])
```

Figure 4. Python code for Tuning as Linear Regression (Bazrafshan et al., 2012)

sion to PRO (Hopkins and May, 2011) which uses linear regression instead of a binary classifier to rank sampled training pairs (briefly: the model is trained to predict the difference in sentence level BLEU scores based on a difference in feature vectors). The complete Python code is given in Fig. 4.

5. Conclusion

We have presented `pycdec`, a high-level Python interface to the fast `cdec` decoder. We illustrated how such an interface allows effortless development of visualizations, training algorithms and applications using machine translation. We hope that the release of our tool will encourage further creative uses of finite-state and context-free methods for machine translation and related applications.

Acknowledgments

This research is supported by the U. S. Army Research Laboratory and the U. S. Army Research Office under contract/grant number W911NF-10-1-0533.

Bibliography

Bazrafshan, M., T. Chung, and D. Gildea. Tuning as linear regression. In *Proc. of NAACL-HLT*, pages 543–547. Association for Computational Linguistics, 2012.

Behnel, S., R. Bradshaw, C. Citro, L. Dalcin, D.S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, March–April 2011.

Bird, S., E. Klein, and E. Loper. *Natural language processing with Python*. O'Reilly Media, 2009. URL `http://nltk.org`.

Chiang, D., Y. Marton, and P. Resnik. Online large-margin training of syntactic and structural translation features. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 224–233. Association for Computational Linguistics, 2008.

Clark, J.H., J. Weese, B.G. Ahn, A. Zollmann, Q. Gao, K. Heafield, and A. Lavie. The machine translation toolpack for LoonyBin: Automated management of experimental machine translation hyperworkflows. *The Prague Bulletin of Mathematical Linguistics*, 93:117–126, 2010.

Dyer, C., J. Weese, H. Setiawan, A. Lopez, F. Ture, V. Eidelman, J. Ganitkevitch, P. Blunsom, and P. Resnik. cdec: A decoder, alignment, and learning framework for finite-state and context-free translation models. In *Proc. of the ACL (Demonstration track)*, pages 7–12. Association for Computational Linguistics, 2010.

Ellson, J., E.R. Gansner, E. Koutsofios, S.C. North, and G. Woodhull. Graphviz and Dynagraph – static and dynamic graph drawing tools. In Junger, M. and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer-Verlag, 2003. URL `http://graphviz.org`.

Federmann, C. and A. Eisele. MT server land: An open-source MT architecture. *The Prague Bulletin of Mathematical Linguistics*, 94:57–66, 2010.

Gimpel, K. and N.A. Smith. Structured ramp loss minimization for machine translation. In *Proceedings of NAACL*, 2012.

Hopkins, M. and J. May. Tuning as ranking. In *Proc. of EMNLP*, pages 1352–1362. Association for Computational Linguistics, 2011.

Koehn, P. An experimental management system. *The Prague Bulletin of Mathematical Linguistics*, 94:87—96, 2010.

Lopez, A. Hierarchical phrase-based translation with suffix arrays. In *Proc. of EMNLP-CoNLL*, pages 976–985, 2007.

Lopez, A. Tera-scale translation models via pattern matching. In *Proc. COLING*, pages 505–512, 2008.

Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. URL `http://scikit-learn.org`.

Pérez, F. and B.E. Granger. IPython: a system for interactive scientific computing. *Comput. Sci. Eng.*, 9(3):21–29, 2007. URL `http://ipython.org`.

Weese, J. and C. Callison-Burch. Visualizing data structures in parsing-based machine translation. *The Prague Bulletin of Mathematical Linguistics*, 93:127–136, 2010.

Address for correspondence:
Victor Chahuneau
`vchahune@cs.cmu.edu`
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA