PYKALDI: A PYTHON WRAPPER FOR KALDI

Doğan Can, Victor R. Martinez, Pavlos Papadopoulos, Shrikanth S. Narayanan

Signal Analysis and Interpretation Lab
University of Southern California
Los Angeles, CA, USA

ABSTRACT

We present PyKaldi, a free and open-source Python wrapper for the widely-used Kaldi speech recognition toolkit. PyKaldi is more than a collection of Python bindings into Kaldi libraries. It is an extensible scripting layer that allows users to work with Kaldi and OpenFst types interactively in Python. It tightly integrates Kaldi vector and matrix types with NumPy arrays. We believe PyKaldi will significantly improve the user experience and simplify the integration of Kaldi into Python workflows. PyKaldi comes with extensive documentation and tests. It is released under the Apache License v2.0 with support for both Python 2.7 and 3.5+.

Index Terms— Speech Recognition, Kaldi, OpenFst, Python

1. INTRODUCTION

Kaldi [1] is a free and open source toolkit for speech recognition. It consists of modern, flexible, general-purpose libraries and executables written in C++ along with a large collection of example scripts for building systems. Since its release in 2011, it has quickly grown to be an indispensable tool for conducting speech research and building speech enabled applications. Kaldi's success should come as no surprise. It has an open license, extensive documentation, tested recipes for building state-of-the-art systems, a large number of contributors from all over the world, a dedicated group of maintainers, and maybe most importantly a well-designed codebase that is easy to understand, modify and extend. Users typically interact with Kaldi either by running its highly modular and composable command-line programs manually inside a UNIX shell or by writing scripts that run these programs. Any functionality that is not exposed by one of the myriad command-line Kaldi programs can be accessed via the C++ application programming interface (API). While this interaction scheme is highly effective, it does not fully address the needs of researchers and developers who would like to use Kaldi in programming languages other than C++.

Thanks to NSF and DARPA for funding. VRM is partially supported by Mexican Council of Science and Technology (CONACyT).

Python is a general-purpose dynamic programming language that is immensely popular in the scientific computing community [2–10]. It has a simple syntax, an extensive standard library, and a mature ecosystem of high quality third-party packages for almost any task including scientific computing[1,2] and machine learning[3,4,5,6]. It comes with one of the best environments for interactive exploration[7], data processing[8], and visualization[9]. Also, the reference CPython implementation exposes a C extension API for implementing new built-in object types and calling into C libraries which can be used to great effect to offload performance critical sections of code to C/C++. For all of these reasons and more, Python bindings for Kaldi libraries is one of the most sought after features among Kaldi users. There exist a number of open source packages[10,11,12,13,14] aiming to bridge the gap between Kaldi and Python, however all of them are fairly limited in scope.

In this paper, we present PyKaldi[15], a free and open source scripting layer (see Figure 1) for Kaldi, that provides a near-complete coverage of Kaldi's C++ library API in Python. PyKaldi is more than a collection of bindings into Kaldi libraries. It provides first class support for Kaldi and OpenFst [11] types to make life easy for the Python users when working with Kaldi. These types can be easily constructed, manipulated, and displayed inside interactive Python interpreters such as IPython [6]. PyKaldi vector and matrix types can be seamlessly converted to NumPy arrays [5] and vice versa by sharing the underlying memory buffers. PyKaldi finite-state transducer (FST) types, including Kaldi style lattices, provide an API similar to the one

[1] http://www.numpy.org
[2] https://www.scipy.org
[3] https://github.com/fchollet/keras
[4] https://github.com/pytorch/pytorch
[5] http://scikit-learn.org
[6] https://www.tensorflow.org
[7] https://ipython.org
[8] http://pandas.pydata.org
[9] http://matplotlib.org
[10] https://github.com/janchorowski/kaldi-python
[11] https://github.com/UFAL-DSG/pykaldi
[12] https://gitlab.idiap.ch/bob/bob.kaldi
[13] https://github.com/yajiemiao/pdnn
[14] https://github.com/srvk/eesen
[15] https://github.com/pykaldi/pykaldi

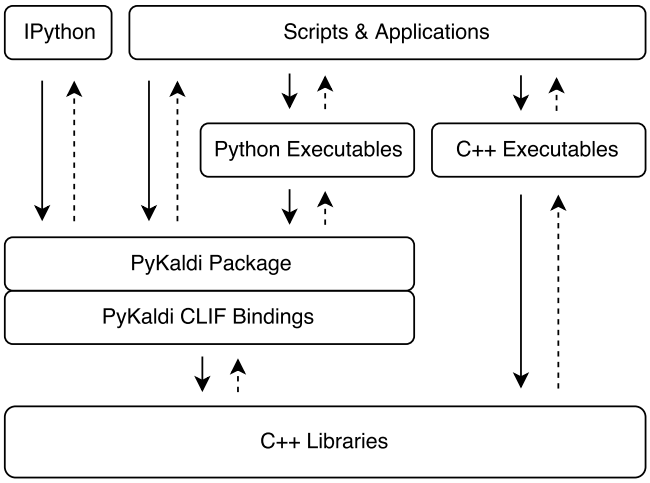

Fig. 1. Extended Kaldi software architecture

provided by OpenFst's official Python wrapper. While it is still in its infancy, PyKaldi already makes working with Kaldi in Python a breeze. Major features of PyKaldi include:

- *Near-complete coverage of Kaldi.*
- *Extensible design.* PyKaldi code is modular and easy to maintain. Kaldi and OpenFst class hierarchies are wrapped at multiple levels exposing generic interfaces in Python. Any changes in Kaldi C++ API can be easily mirrored in Python.
- *Open license.* PyKaldi is licensed under Apache License v2.0.
- *Extensive documentation.* PyKaldi already has extensive documentation for a number of submodules. All API documentation is automatically generated from source files. Also, since most of PyKaldi API directly mirrors Kaldi API, Kaldi's own documentation is applicable to most of PyKaldi.
- *Thorough testing.* PyKaldi already has extensive tests for a number of submodules. These tests are fairly similar to the tests in Kaldi but also have checks for additional Python APIs.
- *Example scripts.* PyKaldi repository includes example Python scripts that are drop in replacements for some Kaldi executables. We are also working on example setups that demonstrate the use of PyKaldi together with popular Python packages.
- *Support for both Python 2.7 and 3.5+.*

2. IMPLEMENTATION

2.1. CLIF bindings

When creating Python bindings for C++ one has a multitude of options. One of the older and more mature projects is Simplified Wrapper and Interface Generator (SWIG)[16]. It connects programs written in C and C++ with multiple languages including Python and Java. Unfortunately, SWIG is infamous for its steep learning curve[17], and the size of the code it generates[18], which makes debugging cumbersome when something goes wrong. Boost.Python[19] and pybind11[20] provide a high-level interface for wrapping C++ code in C++. Both are designed to be non-intrusive, which makes them some of the best options to expose third party libraries in Python. Pybind11 also has dedicated support for NumPy arrays.

Perhaps the most popular option for wrapping C/C++ code in Python is Cython. Cython [8] is both a Python-like language and a compiler. Cython language extends Python with C type annotations for functions, variables and class attributes. It allows the user to write Python code that calls C++ code back and forth natively. When compiled, this generates efficient C code that can be run directly by the CPython interpreter. From a user's perspective, Cython extensions look like any other Python extension. Cython language provides native support for almost all C++ features, including template classes and function overloading.

PyKaldi extension modules are generated with C++ Language Interface Foundation (CLIF)[21]. CLIF is a newly released open-source project developed at Google. CLIF uses LLVM and Clang C++ compiler [12] to parse the type information from a C++ header, which is then used to match the API definition given for that header, and generate an extension module. CLIF allows users to modify the C++ API on-the-fly. This includes renaming classes, functions and methods; mapping functions to Python's magic methods (e.g., constructors to `__init__` or accessors to `__getitem__`); handling class templates and function overloading, treating output parameters as return values, and automatically generating setters and getters for class fields. We selected CLIF because it allowed us to wrap Kaldi's codebase in a concise and readable way. With CLIF, we are able to track Kaldi changes more efficiently, as there are fewer modifications to be made on our side. Moreover, CLIF's auto-generated code is significantly easier to read, understand and modify than the code generated by SWIG or Cython. Unlike Cython, CLIF generated class wrappers do not need to be re-wrapped via composition

[16] www.swig.org
[17] http://www.scipy-lectures.org/advanced/interfacing_with_c/interfacing_with_c.html
[18] https://groups.google.com/forum/#!topic/cython-users/lQO9lGj5JEc
[19] https://github.com/boostorg/python
[20] http://github.com/pybind/pybind11
[21] https://github.com/google/clif

to be directly accessible in CPython. Python objects produced by CLIF can be inherited or directly passed as arguments to wrapped functions without the need for explicit type casting or unwrapping. On the downside, CLIF is a project at its infancy, with limited documentation and examples, and an incipient community. CLIF's main restriction is that it requires code to be C++11 compliant and follow Google C++ style[22], which sometimes forced us to modify or extend Kaldi codebase. For each module, class, and function wrapped, CLIF automatically generates a surrogate *docstring*, a string literal that is used to document the wrapper in Python. To simplify the process of documenting PyKaldi, we extended CLIF's default behavior by allowing optional docstrings to be specified inside CLIF files. When provided, these docstrings are bound to the appropriate documentation slots of the associated Python modules/classes/functions.

2.2. PyKaldi Package

PyKaldi has a modular design which makes it easy to maintain and extend without rebuilding the whole codebase. Source files are organized in a directory tree that is a replica of the Kaldi source tree. Each directory defines a subpackage and contains only the wrapper code written for the associated Kaldi library. The wrapper code consists of:

- CLIF C++ API descriptions defining the types and functions to be wrapped and their Python API,
- C++ headers defining the shims for Kaldi code that is not compliant with the Google C++ style,
- Python modules grouping together related extension modules generated with CLIF and extending the raw CLIF wrappers to provide a more Pythonic API.

While the wrappers generated by CLIF are in most cases adequate for calling into Kaldi libraries, PyKaldi often modifies and extends them in Python (and in some cases in C++) to provide a better user experience. The rest of this section presents good examples of new functionality added by PyKaldi.

2.2.1. Matrix Package

PyKaldi vector and matrix types are tightly integrated with NumPy arrays. They can be easily converted to NumPy arrays and vice versa without copying the underlying memory buffers. They also implement the NumPy array interface which allows them to be used with functions expecting NumPy arrays without explicit conversion. Since conversion to/from Numpy arrays is almost zero-cost, PyKaldi vector and matrix types support the familiar Numpy advanced indexing conventions simply by offloading the `__setitem__` and `__getitem__` operations to NumPy, e.g.

```
v[k] = 2              # set a vector item
m[i,j] = -1           # set a matrix item
m[:,j] = 0            # set a matrix column
m[:,::2] = m[:,1::2]  # set odd matrix columns
```

PyKaldi vectors and matrices can be constructed from other array-like objects. `Vector` and `Matrix` instances are constructed by copying the items from the source objects. `SubVector` and `SubMatrix` instances, on the other hand, share data with the source objects used to construct them whenever possible. A copy is made only if the source object has an `__array__` method and that method returns a copy, or if the source object is a sequence, or if a copy is needed to satisfy any of the other requirements (data type, order, etc.). `SubVector` and `SubMatrix` instances do not own their memory buffers. To make sure their memory buffers are not deallocated while they are still in scope, they keep internal references to objects that they share data with.

2.2.2. FST Package

PyKaldi has built-in support for common FST types (including Kaldi lattices) and operations. The API for the user facing PyKaldi FST types and operations is entirely defined in Python mimicking the API exposed by OpenFst's official Python wrapper to a large extent. This includes integrations with Graphviz [13] and IPython [6] for interactive visualization of FSTs. However, unlike OpenFst's official Python wrapper, which uses Cython, PyKaldi's OpenFst bindings are generated with CLIF so that FST types work seamlessly with the rest of the PyKaldi package. Further, in contrast to OpenFst's official Python wrapper, PyKaldi does not wrap OpenFst scripting API, which uses virtual dispatch, function registration, and dynamic loading of shared objects to provide a common interface shared by FSTs of different semirings. While this change requires wrapping each semiring specialization of an OpenFst class or function template separately, it gives users the ability to pass PyKaldi FST objects directly to the myriad Kaldi functions accepting FST arguments.

2.2.3. Error Handling

Kaldi codebase makes extensive use of assertions for checking the sanity of inputs and self-consistency of computations. Unfortunately, if a Kaldi assertion fails at runtime, it results in an unrecoverable program abort, which is not a great experience to have during an interactive Python session. Further, all Kaldi errors, including assertion failures, print a stack trace which makes it hard to see the actual error message when working interactively. To address these concerns we added new functions to Kaldi that enable/disable stack traces and the abort call in failed assertion handling. PyKaldi disables both by default but the user has the option of enabling them back. In addition to the sanity checks performed by Kaldi, PyKaldi also does its own checks in Python to make sure that the parameters passed to Kaldi have the correct types and sizes.

[22] `http://google.github.io/styleguide/cppguide.html`

3. EXAMPLE

```
# example.py
from kaldi.feat.mfcc import Mfcc, MfccOptions
from kaldi.matrix import SubVector, SubMatrix
from kaldi.util.options import ParseOptions
from kaldi.util.table import SequentialWaveReader
from kaldi.util.table import MatrixWriter
from numpy import mean
from sklearn.preprocessing import scale

usage = """Extract MFCC features.

Usage:  example.py [opts...] <rspec> <wspec>
"""
po = ParseOptions(usage)
po.register_float("min-duration", 0.0,
                  "minimum segment duration")
mfcc_opts = MfccOptions()
mfcc_opts.frame_opts.samp_freq = 8000
mfcc_opts.register(po)

# parse command-line options
opts = po.parse_args()
rspec, wspec = po.get_arg(1), po.get_arg(2)

mfcc = Mfcc(mfcc_opts)
sf = mfcc_opts.frame_opts.samp_freq

with SequentialWaveReader(rspec) as reader, \
     MatrixWriter(wspec) as writer:
    for key, wav in reader:
        if wav.duration < opts.min_duration:
            continue
        assert(wav.samp_freq >= sf)
        assert(wav.samp_freq % sf == 0)
        # >>> print(wav.samp_freq)
        # 16000.0

        s = wav.data()
        # >>> print(s)
        #  11891  28260  ...     360    362
        #  11772  28442  ...     362    414
        # [kaldi.matrix.Matrix of size 2x23001]

        # downsample to sf [default=8kHz]
        s = s[:,::int(wav.samp_freq / sf)]

        # mix-down stereo to mono
        m = SubVector(mean(s, axis=0))

        # compute MFCC features
        f = mfcc.compute_features(m, sf, 1.0)

        # standardize features
        f = SubMatrix(scale(f))
        # >>> print(f)
        # -0.8572 -0.6932  ...   0.5191  0.3885
        # -1.3980 -1.0431  ...   1.4374 -0.2232
        #       ...                    ...
        # -1.7816 -1.4714  ...  -0.0832  0.5536
        # -1.6886 -1.5556  ...   1.0878  1.1813
        # [kaldi.matrix.SubMatrix of size 142x13]

        # write features to archive
        writer[key] = f
```

Listing 1: Extracting MFCC features with PyKaldi

Listing 1 gives an example Python script for extracting MFCC features with PyKaldi and standard utilities from NumPy and scikit-learn [10]. The script first sets the options for MFCC extraction, then iterates over the input table to extract and write MFCC features for each input wave file. Each wave file is downsampled to 8KHz and mixed down to mono before MFCC features are extracted. Raw MFCC features are standardized by removing the mean and scaling to unit variance before they are written out. PyKaldi option parsing API is slightly different from the underlying Kaldi option parsing API. Command-line options for the main script are registered by calling type-specific registration methods that accept name, default value and help string arguments e.g. `min-duration` in the example. The `parse_args` method of a PyKaldi `ParseOptions` instance returns a simple namespace object containing the parsed option values for the main script. Parsed values for other options are directly written into the appropriate fields of associated options instances, e.g. `mfcc_opts` in the example. In typical Kaldi fashion, input/output tables are constructed with read/write specifiers, strings that describe how the data should be read/written. PyKaldi table readers/writers implement the context manager interface, hence they do not need to be closed when they are used in a `with` statement. PyKaldi table writers also support a pseudo-dictionary interface for writing given key value pairs. Since PyKaldi matrices implement NumPy array interface, they can be passed to functions expecting Numpy array arguments, such as `mean` and `scale`, without explicit conversion. The NumPy arrays returned from functions can be easily converted back to Kaldi vector and matrix types by constructing new `SubVector` and `SubMatrix` objects which share the underlying memory buffers with the source arrays whenever possible, i.e. no data is copied unless necessary.

4. CONCLUSION

We described PyKaldi, an open-source Python scripting layer for Kaldi. PyKaldi uses CLIF to generate raw bindings into Kaldi C++ API and extends those bindings in Python to provide a better user experience. At the time of writing, PyKaldi already exposes a large part of the Kaldi C++ API. We believe the next phase of the project will largely focus on adding example setups using PyKaldi together with popular Python packages, making the API more Python friendly, and extending the documentation. We are hopeful that the Kaldi and Python communities will embrace PyKaldi and contribute to its continued development going forward.

5. REFERENCES

[1] Daniel Povey, Arnab Ghoshal, Gilles Boulianne, Lukas Burget, Ondrej Glembek, Nagendra Goel, Mirko Hannemann, Petr Motlicek, Yanmin Qian, Petr Schwarz, Jan Silovsky, Georg Stemmer, and Karel Vesely, "The kaldi

speech recognition toolkit," in *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*. Dec. 2011, IEEE Signal Processing Society, IEEE Catalog No.: CFP11SRW-USB.

[2] Eric Jones, Travis Oliphant, Pearu Peterson, et al., "SciPy: Open source scientific tools for Python," `"http://www.scipy.org/"`, 2001.

[3] Travis E Oliphant, "Python for scientific computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[4] K Jarrod Millman and Michael Aivazis, "Python for scientists and engineers," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 9–12, 2011.

[5] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux, "The numpy array: a structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[6] Fernando Pérez and Brian E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, no. 3, pp. 21–29, May 2007.

[7] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[8] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[9] Wes McKinney, "Data structures for statistical computing in python," in *Proceedings of the 9th Python in Science Conference*. SciPy Austin, TX, 2010, vol. 445, pp. 51–56.

[10] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al., "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[11] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri, "Openfst: A general and efficient weighted finite-state transducer library," *Implementation and Application of Automata*, pp. 11–23, 2007.

[12] Chris Lattner and Vikram Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[13] Emden R. Gansner and Stephen C. North, "An open graph visualization system and its applications to software engineering," *SOFTWARE - PRACTICE AND EXPERIENCE*, vol. 30, no. 11, pp. 1203–1233, 2000.