esig Documentation

Release 0.6

David Maxwell

Aug 28, 2020

Contents

This is the online documentation for the Python `esig` package up to version 0.7.1 - the package is currently undergoing revision to provide substantial additional functionality and speed. The version 0.7.3 only supports Python >= 3.5 and 64bit. It does not support 32 bit installations. It has the same interface with a new functionality in "recombine" that can reduce clouds of paths to a small reweighted subcollection while retaining their expected signature. This code relies on MKL, OpenMP, and heavy parallel linear algebra. As a result the build process from the gz file has changed from this documentation. The wheel, with pip install esig will install on Linux (ManyLinux2014), Mac and Windows but requires a current version of pip (in particular the one shipped with Ubuntu 18.04 is too old - so update it first). If you have trouble please let us know and install 0.7.1. Below you'll find the table of contents for version 0.6.

CHAPTER 1

Prerequisites for the `esig` Package

In order to successfully download, build and run the `esig` package on your computer, you are required to have the following prerequisite software packages installed and correctly configured on your system.

You require:

- Python, version 2.7.x, or version 3.x; and
- the Boost C++ library.

This section provides a brief overview on how to download, setup and configure each of these prerequisites. After completing these steps, `esig` will be able to build correctly on your system (see chap-installer.)

Important: On supported Windows systems, you don't need to install Boost. We do all the hard work for you by creating a series of precompiled Python wheels. If you want to build Boost from source on Windows, you'll need to install Boost and make sure you have the relevant Microsoft Visual C++ compiler.

Warning: The following guides are for reference only. There are no guarantees that the following instructions will work flawlessly on your system, although they *have been tested* and shown to work on a range of systems. For the latest documentation regarding Boost, you should always check out the official Boost documentation page.

1.1 Getting your Python Version

It's a good idea to determine what version of Python you'll be running `esig` on – especially if you are using Windows. **If you're using Linux of macOS, you can skip this section.** This is because you will need to download a version of Boost that will be able to work with your version of Python. To obtain your Python version, open a `Command Prompt` and enter the following command.

```
C:\> python -V
Python 3.6.1
```

The example above shows that the version of Python running is version `3.6.1`. Make a note of this number – you'll need to pick the appropriate Boost downloadable in the following step.

1.2 Installing and Configuring Boost

This section details how you can install and configure (if required) Boost on your system. The process should be straightforward: you should be able to install a precompiled version for your system from your system's package manager (for example, `apt-get` or `yum`). Windows is straightforward, too – although you need to ensure that you have set a special environment variable so that the Boost libraries and header files can be located when you attempt to install `esig`. This guide shows you how to get everything working.

1.2.1 Windows

On Windows, the setup process involves two main steps: *(1)* downloading and installing precompiled Boost libraries; and *(2)* ensuring that your environment variables are correctly configured. We – and our `esig` installer – assume that you are using the default path names for the Boost libraries.

Downloading Boost

There are a large number of precompiled versions of Boost for Windows available online. But which one do you download? **Pick the latest one** – at the time of writing, it is `1.65.1`. Within this directory, there are a variety of different executable downloads. For Boost to work with `esig`, you need to pick a precompiled version of Boost that was compiled with the same Microsoft Visual C compiler as your version of Python. That's why we asked you to get your Python version beforehand – the number will now come in handy.

From the table below, work out what Visual C compiler maps to your version of Python. This table is taken from the official Python documentation – check out this page for more information.

Python Version	Visual C Compiler
2.7, 3.0 - 3.2	`msvc9.0`
3.3 - 3.4	`msvc10.0`
3.5 - 3.6	`msvc14.0`

Once you have worked out what Visual C compiler was used to compile your version of Python on Windows, head back to the Boot precompiled libraries page and select the version you require. In the example screenshot below, the highlighted option `boost_1_65_1-msvc-14.0-64.exe` will provide a 64-bit Boost version `1.65.1` compiled against the `msvc14.0` compiler. Check whether you're using a 32-bit or 64-bit system, too! Today, it's likely you'll be using a 64-bit system.

images/boost-library-select.png

Note: You don't need to actually download the Visual C compiler – we have provided a series of precompiled Python wheels for various versions of Python on Windows. If you however do plan to compile from source, you will of course need to download the appropriate compiler.

Installing Boost

Installing Boost is just like installing any other application on Windows – run the executable installer, and everything will be taken care of for you. The process will take several minutes as there are many files that need to be extracted from the archive.

Once complete, you can check the installed directory. The screenshot below provides an example. Highlighted are the two important directories – the header files are located in `boost`, and the precompiled libraries are present in the other directory.

images/boost-directory.png

Note: Make a note of the directory in which you install Boost to. You'll need this for the next step. The default path is `C:\local\boost_1_65_1\` – replacing the version with the version you have selected. Try to avoid spaces in paths – this makes things easier.

The `BOOST_ROOT` Environment Variable

On Windows machines, the `BOOST_ROOT` environment variable is the recommended way to tell the Visual C compiler where all the Boost libraries and header files live for the version you have installed.

On recent Windows releases (7, 8, 8.1 and 10) you can use the Command Prompt's `setx` tool. Run the following command, replacing `<BOOST_PATH>` with the path to the directory you installed Boost to in the previous step.

```
C:\> setx BOOST_ROOT <BOOST_PATH>

SUCCESS: Specified value was saved.
```

The screenshot below shows the basic process, and also includes an example of using the `set` command to verify that `BOOST_ROOT` has been set correctly.

images/set-env.png

Once you've done this, you are ready to install `esig`.

1.2.2 Linux and macOS

Using your Package Manager

To keep things simple, we highly recommend that you download and install a precompiled version of the Boost libraries for your Linux distribution or macOS system. Depending upon what system you are using, the command you supply to do this somewhat varies.

On macOS, you can either use MacPorts or Homebrew to install the software. With MacPorts, you can try the following command.

```
$ sudo port install boost
```

Alternatively, if you have Homebrew installed, try this command.

```
$ sudo brew install boost
```

Both should install Boost without problem to a default path, and from here, you're good to go.

Linux commands are pretty similar. If you're using Ubuntu try the following command.

```
$ sudo apt-get install libboost-all-dev
```

Alternatively, a Red Hat based system will use `yum` – the following command should work on Fedora, CentOS, RHEL and other Red Hat-based systems.

```
$ sudo yum install boost-devel
```

If your distribution isn't listed here, then a quick Web search should provide you with all the information that you need.

Building from Source

If your system doesn't have a package manager, or it doesn't provide Boost, you can always download the Boost sources and compile them yourself. Download the Boost sources for UNIX from here (the `.tar.gz` file), and extract everything to a temporary directory.

Warning: You need to ensure that you have all the development tools your system needs to compile the Boost libraries. For macOS, this will involve installing Xcode. On Linux distributions, this will involve installing the necessary packages (e.g. `sudo apt-get install build-essential` or `sudo yum groupinstall "Development Tools"`.)

After everything has been extracted, open a terminal and `cd` to the extracted directory. The following commands must then be run.

```
$ ./bootstrap.sh
$ sudo ./b2 install
```

Running the second command requires `sudo` privileges as it compiles the software and then attempts to copy the Boost header files and compiled libraries to `/usr/local/`, which is the default path for the installation of such components. If you don't have `sudo` access, you can always compile and install the components to a directory within your home directory with the following commands.

```
$ ./bootstrap.sh --prefix=$HOME/local/
$ ./b2 install
```

The example above will install Boost to `$HOME$/local/`, where `$HOME` represents the path to your home directory. If you go down this custom path (no pun intended), you'll need to make sure that the installer can see the necessary header files and libraries, otherwise compilation of `esig` **will fail.** Refer to *Installing esig* to see how to do this. When running `esig` this way, you'll need to make sure you have set the `LD_LIBRARY_PATH` (or `DYLD_LIBRARY_PATH` on macOS) environment variable to also point to where you installed the compiled Boost libraries.

Assuming that Boost libraries are installed to `$HOME/local/boost/lib/`, you can set the environment variable as shown in the example below.

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/local/boost/lib
```

On macOS, change `LD_LIBRARY_PATH` to `DYLD_LIBRARY_PATH`. You can place this command in your `~/.profile` or `~/.bashrc` files to ensure that this path is set everytime you start a new terminal.

CHAPTER 2

Installing `esig`

Once you have checked out the *Prerequisites for the esig Package*, you're ready to install `esig`. And it should be really easy! To install, create a new virtual environment (if you want to), or activate the one you wish to install it to. Once you're ready to install, run the following command from your terminal or Windows command prompt.

```
$ pip install esig
```

That should be it. `esig` should install, either from a precompiled wheel for your platform, or build the package from the source. If it builds from source, expect compilation time to take 5-10 minutes depending upon your system. If you already have `esig` installed and wish to upgrade to a newer version, run the following command.

```
$ pip install esig --upgrade
```

Note: If building from source (i.e. the `.tar.gz` archive), note that the terminal may seem unresponsive. Don't worry, it's compiling – it just takes a few minutes.

Once installed, you can test that the install process worked as expected by trying the following commands.

```
$ python -c "import esig; esig.is_library_loaded()"
True
```

If you see `True`, then all is well, and you're ready to go. If you don't see `True`, but `False` and an error, check out *Troubleshooting the esig Installation*.

2.1 Custom Library and Include Paths

If you require to build `esig` from source and have installed Boost or any other prerequisite to a non-standard location, we've provided functionality for you to specify where the installer should look for the relevant header files and libraries. This functionality is provided by supplying to additional command-line arguments to the installation scripts for `esig` – `include-dirs` and `library-dirs`. More than one directory can be supplied for each argument, separated by the path separator character for your platform (`;` for Windows, `:` for other platforms). You don't need to supply both arguments; if for example you only need to supply a path to libraries, you only need to supply the `library-dirs` argument.

To supply these arguments to `pip`, you need to wrap them up inside an `install-option` parameter. For example, if we have include files located at `/opt/boost/include` and libraries located at both `/opt/boost/lib` and `/opt/other/lib` on a Linux installation, we would supply the following command.

```
$ pip install esig --install-option="--include-dirs=/opt/boost/include" --install-
↪option="--library-dirs=/opt/boost/lib:/opt/other/lib"
```

Note each argument needs to be wrapped inside its own `--install-option` parameter. If installing directly from your local filesystem, just call the `setup.py` module like so.

```
$ python setup.py --include-dirs=/opt/boost/include --library-dirs=/opt/boost/lib:/
↪opt/other/lib
```

Your additional paths are added to the two lists of search directories, so everything should be able to be found and used as required during the build process.

2.2 Building/Installing Fails!

The `pip` package manager is designed to keep things as simple as possible to the user. As such, this means keeping output on the user's terminal to a minimum. While this is fine for 99% of scenarios, when things go wrong it's more useful to have as much information at your disposal. `pip` by default suppresses the output from installation scripts that it runs, meaning that it can be difficult to work out what goes wrong.

If you are building `esig` from source and find that the build fails, the specially-crafted `esig` installer will provide some useful output on lines starting with `esig_installer`. To access this output, run the installer again with the `-v` switch (`v` for verbose). As an example, your command will be `$ pip install esig -v`. From this information, you'll be able to view *Troubleshooting the esig Installation* with more of an idea of what is wrong.

2.3 Running Tests

Once you have installed the software, we recommend that you run the provided unit tests. Doing so will give you confidence that the software is returning correct output before you begin your experimentation.

To run tests, start your `python` interpreter, and follow the example below.

```
Python 3.6.1 (default, Jun 29 2017, 15:17:57)
[GCC 4.2.1 Compatible Apple LLVM 8.1.0 (clang-802.0.42)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from esig import tests
>>> tests.run_tests()
.......
----------------------------------------------------------------------
Ran 7 tests in 0.033s

OK
>>>
```

After calling `run_tests()`, you should see all tests pass (at the time of writing, seven tests were implemented). If a test(s) fail(s), you should contact us with information on your platform (operating system used, Python version used) and what test(s) fail(s) – you may have discovered a bug that needs to be patched.

CHAPTER 3

Troubleshooting the `esig` Installation

If you find yourself on this page, you may well have been directed to visit by the installer. Here, we detail and provide solutions to a number of commonly occurring problems that you may find when attempting to install and use the `esig` package. Compiling and ensuring that `esig` works correctly on a wide range of platforms is not trivial – a lot of work has gone into ensuring it works on the greatest number of systems as possible. However, bad things can happen. Hopefully this page will be able to resolve your problem – if not, feel free to contact one of the team listed on *The esig Python Package*.

3.1 Unknown Command `bdist_wheel`

When compiling from source, you may find that the installer fails stating that the command `bdist_wheel` is not a valid command. This is because your `setuptools` package is out of date, and/or you do not have the `wheel` package installed.

To fix this problem, run the following commands. If you are using virtual environments, ensure you have activated your virtual environments beforehand.

```
$ pip install setuptools --upgrade
$ pip install wheel --upgrade
```

These commands should fix the problem, and `esig` should then install without problem.

3.2 Permission Denied when Installing

When installing `esig`, you recieve a `Permission Denied` error. This means that copying package files to the appropriate location failed as your account didn't have sufficient privileges to do so. This will happen when you attempt to install `esig` globally for your Python installation. You can do this by running the `pip install esig` command with elevated privileges (e.g. `sudo pip install esig` on macOS/Linux, or running the command in a Windows Command Prompt with elevated privileges). However, we recommened that you use Python virtual environments, and install `esig` to one of those.

3.3 Can't Load Boost libraries

If you attempt to import `esig` when running Python and you find an error stating that certain libraries cannot be imported, you've most likely installed libraries that `esig` are dependent upon in a non-standard location. The solution to this problem is to add the paths to the required libraries to your `LD_LIBRARY_PATH` or `DYLD_LIBRARY_PATH` environment variable, on Linux or macOS respectively. Windows users will not run into this problem.

3.4 `numpy` error

We're working on finding a solution for this problem.

CHAPTER 4

`esig`: What is it?

The `esig` package provides implementations of a series of mathematical tools for the handling of the *signature* of a *path*. Powered by the underlying `libalgebra` C++ package, developed by Terry Lyons et al. over a period of 15 years, the `esig` package has been further developed and extended by three 2017 Summer interns at the Alan Turing Institute, London, England. Below we provide an explanation of what a path and a signature are – it gets mathematical.

Consider a *path* as smooth mapping from a time interval into some multidimensional space; it is a special case of a continuous stream of data. A *signature* is a canonical transform of a data stream into a high dimensional space of features; `esig` is a package for implementing this transform for paths – as a mathematical object, a signature is an infinite tensor series. `esig` works with the truncated signatures having a certain *degree*. Formally and for piecewize linear or smooth paths, each coordinate of the signature tensor is an iterated Riemann-Stieltjes integral; the ensemble of these values represents an element in the tensor algebra of Euclidean space. Precise definitions and an introduction to this particular research area can be found in this article written by Terry Lyons, as well as this survey paper.

In practice, paths are easy to visualize and summarise in the case where they are (well approximated) by piecewise linear functions. If the initial and end points, as well as the intermediate points where direction changes, are collected into an array then this discrete stream contains a full description of the path. The `esig` package takes such an array as argument, and computes the signature (or logsignature) of the associated path truncated at a chosen degree.

The `esig` package is a python package, and the function stream2sig requires an input array that is an np.array((l,n),dtype=float) of shape `(l,n)` where: `l` is the number of sample points in the path including both ends and `n` is the dimension of the target space where the points live. In the following example, the path is given as the list of coordinate vectors of points defining the piecewise linear path.

```
(0,1) --> (1,1) --> (2,2) --> (3,0)
```

The signature is then computed by calling a degree, after formatting the data as a float based `numpy.array`. The degree is roughly defined as the number of iterations in the integration performed. In this example, degree 3 means that all of the possible combinations of the coordinate functions

```
x_1 = (0 --> 1 --> 2 --> 3)
x_2 = (1 --> 1 --> 2 --> 0)
```

of length 3 or lower are integrated over. There is 1 combination of length 0, 2 combinations of lenght 1, 4 combinations of length 2, 8 combinations of length 3, making the resulting vector have dimension 15.

The signature is a vectorisation of the path where the dimension of the feature set does not depend on the number of points along the path.

If you're interested in how the software was packaged up, you can look at the small reflection paper written by one of the Summer interns, David Maxwell. The paper is available on arXiv.

CHAPTER 5

`esig`: How to use it?

Once installed the `esig` package provides a very simple implementations for a limited series of core mathematical operations for the handling of the *signature* of a *path.*

To get started import tosig from esig:

```
>>> from esig import tosig as ts
```

Then get help on the functionality tosig provides

```
>>> help(ts)
```

NAME esig.tosig - This is the tosig module from ESIG

FUNCTIONS

logsigdim(...) logsigdim(signal_dimension, signature_degree) returns a Py_ssize_t integer giving the dimension of the log signature vector returned by stream2logsig

logsigkeys(...) logsigkeys(signal_dimension, signature_degree) returns, in the order used by ...2logsig, a space separated ascii string containing the keys associated the entries in the log signature returned by ...2logsig

sigdim(...) sigdim(signal_dimension, signature_degree) returns a Py_ssize_t integer giving the length of the signature vector returned by stream2logsig

sigkeys(...) sigkeys(signal_dimension, signature_degree) returns, in the order used by ...2sig, a space separated ascii string containing the keys associated the entries in the signature returned by ...2sig

stream2logsig(...) stream2logsig(array(no_of_ticks x signal_dimension), signature_degree) reads a 2 dimensional numpy array of floats, “the data in stream space” and returns a numpy vector containing the logsignature of the vector series up to given signature_degree

stream2sig(...) stream2logsig(array(no_of_ticks x signal_dimension), signature_degree) reads a 2 dimensional numpy array of floats, “the data in stream space” and returns a numpy vector containing the signature of the vector series up to given signature_degree

To understand better, lets look at some of the helper functions

```
>>> ts.sigkeys(2,4)
' () (1) (2) (1,1) (1,2) (2,1) (2,2) (1,1,1) (1,1,2) (1,2,1) (1,2,2) (2,1,1) (2,1,2)␣
↪(2,2,1) (2,2,2) (1,1,1,1) (1,1,1,2) (1,1,2,1) (1,1,2,2) (1,2,1,1) (1,2,1,2) (1,2,2,
↪1) (1,2,2,2) (2,1,1,1) (2,1,1,2) (2,1,2,1) (2,1,2,2) (2,2,1,1) (2,2,1,2) (2,2,2,1)␣
↪(2,2,2,2)'
```

enumerates a basis for the tensor algebra on an alphabet of size 2 up to degree 4 as a text string. ts.sigkeys(2,4).strip().split(" ") provides the same basis as a list. The signature is then computed by calling stream2sig(path_data, truncation_degree), after formatting path_data as a `numpy.array`. The degree d is roughly defined as the maximal order to which iterated integration is performed. The signature can then be represented as a linear combination of words of length at most d. The logsignature has a more complicated but more compact representation.

`esig` has other helper functions that can be used to access the data effectively.

```
    >>> ts.sigkeys(2,4).strip().split(" ")[ts.sigdim(2,2):ts.sigdim(2,3)]
['(1,1,1)', '(1,1,2)', '(1,2,1)', '(1,2,2)', '(2,1,1)', '(2,1,2)', '(2,2,1)', '(2,2,2)
↪']
```

trims the full basis to only those elements that have degree three in the tensor basis.

Very similar code works for calculating, accessing and manipulating the basis for the lie elements used for log signatures and the log signature itself:

```
>>> ts.logsigkeys(2,4).strip().split(" ")[ts.logsigdim(2,2):ts.logsigdim(2,3)]
['[1,[1,2]]', '[2,[1,2]]']
```

Computing signatures and log signatures from the discrete sequences is achieved using stream2sig and stream2logsig.

Following the discussion above: The input stream is a numpy array of floats (not integers!); if one varies the first index and fixes the second index then the values are those along a coordinate path (not increments), the number of coordinate paths or channels in the path is determined by the second index, stream2sig deduces this from the width of the numpy array. The depth of the calculated signature is an argument to be chosen by the user. The ouput is a one dimensional array of numbers and is to be interpreted as the coefficients of the basis elements produced by sigkeys delivered in the same order.

The program as written will only do calculations that fit easily into 32 bit memory and will reject widths and depths that produce larger problems with an error message at the command line. This is to discourage exponentially large problems that hang your computer. The underlying C++ code is structured as a sparse tensor. It can handle much larger problems. The source code is available in the libalgebra folder in the PyPi source code file (the gz file).

The stream2… code is thread safe and parallelizing it can dramatically accelerate computations where many signatures need to be computed. All signature code works through massive memoisation so this thread safety should never be assumed without testing.

CHAPTER 6

Who is Involved?

Over the Summer of 2017, a team of interns worked on developing the `esig` package to a point where it was ready to be shown to the world. The interns were working at the Alan Turing Institute, London, England. They were supervised by Professor Terry Lyons of the Mathematical Institute, Oxford, and Dr Hao Ni of UCL.

We should note that the the underlying C++ libraries that allow `esig` to exist have been developed by Professor Lyons and his colleagues and students over a number of years leading up to this internship. Without their hard work, this project would not exist as it is today. In a future revision of this documentation, we'll update this portion with the names of those who were involved.

For now though, here are the five people who have worked on developing `esig` in recent times.