Joblib: running Python functions as pipeline jobs

Introduction

Joblib is a set of tools to provide **lightweight pipelining in Python**. In particular:

1. transparent disk-caching of functions and lazy re-evaluation (memoize pattern)
2. easy simple parallel computing

Joblib is optimized to be **fast** and **robust** on large data in particular and has specific optimizations for *numpy* arrays. It is **BSD-licensed**.

Documentation:	https://joblib.readthedocs.io
Download:	https://pypi.python.org/pypi/joblib#downloads
Source code:	https://github.com/joblib/joblib
Report issues:	https://github.com/joblib/joblib/issues

Vision

The vision is to provide tools to easily achieve better performance and reproducibility when working with long running jobs.

- **Avoid computing the same thing twice**: code is often rerun again and again, for instance when prototyping computational-heavy jobs (as in scientific development), but hand-crafted solutions to alleviate this issue are error-prone and often lead to unreproducible results.
- **Persist to disk transparently**: efficiently persisting arbitrary objects containing large data is hard. Using joblib's caching mechanism avoids hand-written persistence and implicitly links the file on disk to the execution context of the original Python object. As a result, joblib's persistence is good for resuming an application status or computational job, eg after a crash.

Joblib addresses these problems while **leaving your code and your flow control as unmodified as possible** (no framework, no new paradigms).

Main features

1. **Transparent and fast disk-caching of output value:** a memoize or make-like functionality for Python functions that works well for arbitrary Python objects, including very large numpy arrays. Separate persistence and flow-execution logic from domain logic or algorithmic code by writing the operations as a set of steps with well-defined inputs and outputs: Python functions. Joblib can save their computation to disk and rerun it only if necessary:

```
>>> from joblib import Memory
>>> location = 'your_cache_dir_goes_here'
>>> mem = Memory(location, verbose=1)
>>> import numpy as np
>>> a = np.vander(np.arange(3)).astype(float)
>>> square = mem.cache(np.square)
>>> b = square(a)
________________________________________________________________________...
[Memory] Calling ...square...
square(array([[0., 0., 1.],
       [1., 1., 1.],
       [4., 2., 1.]]))
_____________________________________________...square - ...s, 0.0min
```

```
>>> c = square(a)
>>> # The above call did not trigger an evaluation
```

2. **Embarrassingly parallel helper:** to make it easy to write readable parallel code and debug it quickly:

```
>>> from joblib import Parallel, delayed
>>> from math import sqrt
>>> Parallel(n_jobs=1)(delayed(sqrt)(i**2) for i in range(10))
[0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]
```

3. **Fast compressed Persistence**: a replacement for pickle to work efficiently on Python objects containing large data (*joblib.dump* & *joblib.load*).

User manual

Why joblib: project goals

- Benefits of pipelines
- Joblib's approach
- Design choices

Installing joblib

- Using *pip*
- Using distributions
- The manual way

On demand recomputing: the *Memory* class

- Use case
- Using with *numpy*
- Shelving: using references to cached values
- Gotchas
- Ignoring some arguments
- Custom cache validation
- Reference documentation of the `Memory` class
- Useful methods of decorated functions

Embarrassingly parallel for loops

- Common usage
- Thread-based parallelism vs process-based parallelism
- Setting up `joblib`'s backend with `parallel_config`
- Serialization & Processes
- Shared-memory semantics
- Reusing a pool of workers
- Working with numerical data in shared memory (memmapping)
- Avoiding over-subscription of CPU resources
- Old multiprocessing backend

Module reference

Memory([location, backend, mmap_mode, ...])	A context object for caching a function's return value each time it is called with the same input arguments.
Parallel([n_jobs, backend, return_as, ...])	Helper class for readable parallel mapping.
parallel_config([backend, n_jobs, verbose, ...])	Set the default backend or configuration for **Parallel**.

cpu_count([only_physical_cores])	Return the number of CPUs.
dump(value, filename[, compress, protocol])	Persist an arbitrary Python object into one file.
load(filename[, mmap_mode, ...])	Reconstruct a Python object from a file persisted with joblib.dump.

hash(obj[, hash_name, coerce_mmap])	Quick calculation of a hash to identify uniquely Python objects containing numpy arrays.
register_compressor(compressor_name, compressor)	Register a new compressor.

Deprecated functionalities

parallel_backend(backend[, n_jobs, ...])	Change the default backend used by Parallel inside a with block.

Find out how Algolia AI Search can instantly and precisely understand your user's intent. **Watch Demo**

stable

Ads by EthicalAds