

# How to optimize for speed

The following gives some practical guidelines to help you write efficient code for the scikit-learn project.

## Note

While it is always useful to profile your code so as to **check performance assumptions**, it is also highly recommended to **review the literature** to ensure that the implemented algorithm is the state of the art for the task before investing into costly implementation optimization.

Times and times, hours of efforts invested in optimizing complicated implementation details have been rendered irrelevant by the subsequent discovery of simple **algorithmic tricks**, or by using another algorithm altogether that is better suited to the problem.

The section [ref: warm-restarts](#) gives an example of such a trick.

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 23); [backlink](#)

Unknown interpreted text role "ref".

## Python, Cython or C/C++?

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 29)

Unknown directive type "currentmodule".

```
.. currentmodule:: sklearn
```

In general, the scikit-learn project emphasizes the **readability** of the source code to make it easy for the project users to dive into the source code so as to understand how the algorithm behaves on their data but also for ease of maintainability (by the developers).

When implementing a new algorithm it is thus recommended to **start implementing it in Python using Numpy and Scipy** by taking care of avoiding looping code using the vectorized idioms of those libraries. In practice this means trying to **replace any nested for loops by calls to equivalent Numpy array methods**. The goal is to avoid the CPU wasting time in the Python interpreter rather than crunching numbers to fit your statistical model. It's generally a good idea to consider NumPy and SciPy performance tips:

<https://scipy.github.io/old-wiki/pages/PerformanceTips>

Sometimes however an algorithm cannot be expressed efficiently in simple vectorized Numpy code. In this case, the recommended strategy is the following:

1. **Profile** the Python implementation to find the main bottleneck and isolate it in a **dedicated module level function**. This function will be reimplemented as a compiled extension module.
2. If there exists a well maintained BSD or MIT C/C++ implementation of the same algorithm that is not too big, you can write a **Cython wrapper** for it and include a copy of the source code of the library in the scikit-learn source tree: this strategy is used for the classes `:class:'svm.LinearSVC'`, `:class:'svm.SVC'` and `:class:'linear_model.LogisticRegression'` (wrappers for liblinear and libsvm).

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 53); [backlink](#)

Unknown interpreted text role "class".

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 53); [backlink](#)

Unknown interpreted text role "class".

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 53); [backlink](#)

Unknown interpreted text role "class".

3. Otherwise, write an optimized version of your Python function using **Cython** directly. This strategy is used for the `:class:'linear_model.ElasticNet'` and `:class:'linear_model.SGDClassifier'` classes for instance.

**System Message: ERROR/3** (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 61); [backlink](#)

Unknown interpreted text role "class".

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 61); [backlink](#)**

Unknown interpreted text role "class".

4. **Move the Python version of the function in the tests** and use it to check that the results of the compiled extension are consistent with the gold standard, easy to debug Python version.
5. Once the code is optimized (not simple bottleneck spottable by profiling), check whether it is possible to have **coarse grained parallelism** that is amenable to **multi-processing** by using the `joblib.Parallel` class.

When using Cython, use either

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 77)**

Unknown directive type "prompt".

```
.. prompt:: bash $  
  
python setup.py build_ext -i  
python setup.py install
```

to generate C files. You are responsible for adding .c/.cpp extensions along with build parameters in each submodule `setup.py`.

C/C++ generated files are embedded in distributed stable packages. The goal is to make it possible to install scikit-learn stable version on any machine with Python, Numpy, Scipy and C/C++ compiler.

## Profiling Python code

In order to profile Python code we recommend to write a script that loads and prepare you data and then use the IPython integrated profiler for interactively exploring the relevant part for the code.

Suppose we want to profile the Non Negative Matrix Factorization module of scikit-learn. Let us setup a new IPython session and load the digits dataset and as in the [ref: sphx\\_glr\\_auto\\_examples\\_classification\\_plot\\_digits\\_classification.py](#) example:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 98); [backlink](#)**

Unknown interpreted text role "ref".

```
In [1]: from sklearn.decomposition import NMF  
  
In [2]: from sklearn.datasets import load_digits  
  
In [3]: X, _ = load_digits(return_X_y=True)
```

Before starting the profiling session and engaging in tentative optimization iterations, it is important to measure the total execution time of the function we want to optimize without any kind of profiler overhead and save it somewhere for later reference:

```
In [4]: %timeit NMF(n_components=16, tol=1e-2).fit(X)  
1 loops, best of 3: 1.7 s per loop
```

To have a look at the overall performance profile using the `%prun` magic command:

```
In [5]: %prun -l nmf.py NMF(n_components=16, tol=1e-2).fit(X)  
14496 function calls in 1.682 CPU seconds
```

Ordered by: internal time  
List reduced from 90 to 9 due to restriction <'nmf.py'>

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
36	0.609	0.017	1.499	0.042	nmf.py:151(_nls_subproblem)
1263	0.157	0.000	0.157	0.000	nmf.py:18(pos)
1	0.053	0.053	1.681	1.681	nmf.py:352(fit_transform)
673	0.008	0.000	0.057	0.000	nmf.py:28(norm)
1	0.006	0.006	0.047	0.047	nmf.py:42(initialize_nmf)
36	0.001	0.000	0.010	0.000	nmf.py:36(sparseness)
30	0.001	0.000	0.001	0.000	nmf.py:23(neg)
1	0.000	0.000	0.000	0.000	nmf.py:337(__init__)
1	0.000	0.000	1.681	1.681	nmf.py:461(fit)

The `totttime` column is the most interesting: it gives to total time spent executing the code of a given function ignoring the time spent in executing the sub-functions. The real total time (local code + sub-function calls) is given by the `cumtime` column.

Note the use of the `-l nmf.py` that restricts the output to lines that contains the "nmf.py" string. This is useful to have a quick look at the hotspot of the nmf Python module it-self ignoring anything else.

Here is the beginning of the output of the same command without the `-l nmf.py` filter:

```
In [5]: %prun NMF(n_components=16, tol=1e-2).fit(X)  
16159 function calls in 1.840 CPU seconds
```

Ordered by: internal time

ncalls	totttime	percall	cumtime	percall	filename:lineno(function)
2833	0.653	0.000	0.653	0.000	{numpy.core._dotblas.dot}

```

46      0.651    0.014    1.636    0.036 nmf.py:151(_nls_subproblem)
1397    0.171    0.000    0.171    0.000 nmf.py:18(_pos)
2780    0.167    0.000    0.167    0.000 {method 'sum' of 'numpy.ndarray' objects}
1      0.064    0.064    1.840    1.840 nmf.py:352(fit_transform)
1542    0.043    0.000    0.043    0.000 {method 'flatten' of 'numpy.ndarray' objects}
337     0.019    0.000    0.019    0.000 {method 'all' of 'numpy.ndarray' objects}
2734    0.011    0.000    0.181    0.000 fromnumeric.py:1185(sum)
2       0.010    0.005    0.010    0.005 {numpy.linalg.lapack_lite.dgesdd}
748     0.009    0.000    0.065    0.000 nmf.py:28(norm)
...

```

The above results show that the execution is largely dominated by dot products operations (delegated to blas). Hence there is probably no huge gain to expect by rewriting this code in Cython or C/C++: in this case out of the 1.7s total execution time, almost 0.7s are spent in compiled code we can consider optimal. By rewriting the rest of the Python code and assuming we could achieve a 1000% boost on this portion (which is highly unlikely given the shallowness of the Python loops), we would not gain more than a 2.4x speed-up globally.

Hence major improvements can only be achieved by **algorithmic improvements** in this particular example (e.g. trying to find operation that are both costly and useless to avoid computing then rather than trying to optimize their implementation).

It is however still interesting to check what's happening inside the `_nls_subproblem` function which is the hotspot if we only consider Python code: it takes around 100% of the accumulated time of the module. In order to better understand the profile of this specific function, let us install `line_profiler` and wire it to IPython:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 186)**

Unknown directive type "prompt".

```

.. prompt:: bash $

    pip install line_profiler

```

- Under IPython 0.13+, first create a configuration profile:

**System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 192)**

Unknown directive type "prompt".

```

.. prompt:: bash $

    ipython profile create

```

Then register the `line_profiler` extension in `~/.ipython/profile_default/ipython_config.py`:

```

c.TerminalIPythonApp.extensions.append('line_profiler')
c.InteractiveShellApp.extensions.append('line_profiler')

```

This will register the `%lprun` magic command in the IPython terminal application and the other frontends such as qtconsole and notebook.

Now restart IPython and let us use this new toy:

```

In [1]: from sklearn.datasets import load_digits

In [2]: from sklearn.decomposition import NMF
... : from sklearn.decomposition._nmf import _nls_subproblem

In [3]: X, _ = load_digits(return_X_y=True)

In [4]: %lprun -f _nls_subproblem NMF(n_components=16, tol=1e-2).fit(X)
Timer unit: 1e-06 s

File: sklearn/decomposition/nmf.py
Function: _nls_subproblem at line 137
Total time: 1.73153 s

```

Line #	Hits	Time	Per Hit	% Time	Line Contents
137					def _nls_subproblem(V, W, H_init, tol, max_iter):
138					"""Non-negative least square solver
...					
170					"""
171	48	5863	122.1	0.3	if (H_init < 0).any():
172					raise ValueError("Negative values in H_init passed to NLS")
173					
174	48	139	2.9	0.0	H = H_init
175	48	112141	2336.3	5.8	WtV = np.dot(W.T, V)
176	48	16144	336.3	0.8	WtW = np.dot(W.T, W)
177					
178					# values justified in the paper
179	48	144	3.0	0.0	alpha = 1
180	48	113	2.4	0.0	beta = 0.1
181	638	1880	2.9	0.1	for n_iter in range(1, max_iter + 1):
182	638	195133	305.9	10.2	grad = np.dot(WtW, H) - WtV
183	638	495761	777.1	25.9	proj_gradient = norm(grad[np.logical_or(grad < 0, H > 0)])
184	638	2449	3.8	0.1	if proj_gradient < tol:
185	48	130	2.7	0.0	break
186					
187	1474	4474	3.0	0.2	for inner_iter in range(1, 20):

188	1474	83833	56.9	4.4	Hn = H - alpha * grad
189					# Hn = np.where(Hn > 0, Hn, 0)
190	1474	194239	131.8	10.1	Hn = _pos(Hn)
191	1474	48858	33.1	2.5	d = Hn - H
192	1474	150407	102.0	7.8	gradd = np.sum(grad * d)
193	1474	515390	349.7	26.9	dQd = np.sum(np.dot(WtW, d) * d)
...					

By looking at the top values of the % Time column it is really easy to pin-point the most expensive expressions that would deserve additional care.

## Memory usage profiling

You can analyze in detail the memory usage of any Python code with the help of [memory\\_profiler](#). First, install the latest version:

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 262)
Unknown directive type "prompt".

.. prompt:: bash $

    pip install -U memory_profiler
```

Then, setup the magics in a manner similar to `line_profiler`.

- Under IPython 0.11+, first create a configuration profile:

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 270)
Unknown directive type "prompt".

.. prompt:: bash $

    ipython profile create
```

Then register the extension in `~/.ipython/profile_default/ipython_config.py` alongside the line profiler:

```
c.TerminalIPythonApp.extensions.append('memory_profiler')
c.InteractiveShellApp.extensions.append('memory_profiler')
```

This will register the `%memit` and `%mprun` magic commands in the IPython terminal application and the other frontends such as qtconsole and notebook.

`%mprun` is useful to examine, line-by-line, the memory usage of key functions in your program. It is very similar to `%lprun`, discussed in the previous section. For example, from the `memory_profiler` examples directory:

```
In [1] from example import my_func

In [2] %mprun -f my_func my_func()
Filename: example.py

Line #      Mem usage  Increment  Line Contents
=====
      3                      @profile
      4      5.97 MB    0.00 MB    def my_func():
      5     13.61 MB    7.64 MB        a = [1] * (10 ** 6)
      6     166.20 MB  152.59 MB        b = [2] * (2 * 10 ** 7)
      7     13.61 MB -152.59 MB        del b
      8     13.61 MB    0.00 MB        return a
```

Another useful magic that `memory_profiler` defines is `%memit`, which is analogous to `%timeit`. It can be used as follows:

```
In [1]: import numpy as np

In [2]: %memit np.zeros(1e7)
maximum of 3: 76.402344 MB per loop
```

For more details, see the docstrings of the magics, using `%memit?` and `%mprun?`.

## Performance tips for the Cython developer

If profiling of the Python code reveals that the Python interpreter overhead is larger by one order of magnitude or more than the cost of the actual numerical computation (e.g. `for` loops over vector components, nested evaluation of conditional expression, scalar arithmetic...), it is probably adequate to extract the hotspot portion of the code as a standalone function in a `.pyx` file, add static type declarations and then use Cython to generate a C program suitable to be compiled as a Python extension module.

The official documentation available at <http://docs.cython.org/> contains a tutorial and reference guide for developing such a module. In the following we will just highlight a couple of tricks that we found important in practice on the existing cython codebase in the scikit-learn project.

TODO: html report, type declarations, bound checks, division by zero checks, memory alignment, direct blas calls...

- <https://www.youtube.com/watch?v=gMvkiQ-gOW8>
- [http://conference.scipy.org/proceedings/SciPy2009/paper\\_1/](http://conference.scipy.org/proceedings/SciPy2009/paper_1/)
- [http://conference.scipy.org/proceedings/SciPy2009/paper\\_2/](http://conference.scipy.org/proceedings/SciPy2009/paper_2/)

## Using OpenMP

Since scikit-learn can be built without OpenMP, it's necessary to protect each direct call to OpenMP. This can be done using the following syntax:

```
# importing OpenMP
IF SKLEARN_OPENMP_PARALLELISM_ENABLED:
    cimport openmp

# calling OpenMP
IF SKLEARN_OPENMP_PARALLELISM_ENABLED:
    max_threads = openmp.omp_get_max_threads()
ELSE:
    max_threads = 1
```

### Note

Protecting the parallel loop, `prange`, is already done by cython.

## Profiling compiled extensions

When working with compiled extensions (written in C/C++ with a wrapper or directly as Cython extension), the default Python profiler is useless: we need a dedicated tool to introspect what's happening inside the compiled extension it-self.

## Using yep and gperftools

Easy profiling without special compilation options use yep:

- <https://pypi.org/project/yep/>
- <http://fa.bianp.net/blog/2011/a-profiler-for-python-extensions>

## Using gprof

In order to profile compiled Python extensions one could use `gprof` after having recompiled the project with `gcc -pg` and using the `python-dbg` variant of the interpreter on debian / ubuntu: however this approach requires to also have `numpy` and `scipy` recompiled with `-pg` which is rather complicated to get working.

Fortunately there exist two alternative profilers that don't require you to recompile everything.

## Using valgrind / callgrind / kcachegrind

### kcachegrind

`yep` can be used to create a profiling report. `kcachegrind` provides a graphical environment to visualize this report:

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 402
```

Unknown directive type "prompt".

```
.. prompt:: bash $

# Run yep to profile some python script
python -m yep -c my_file.py
```

```
System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\developers\scikit-learn-main) (doc) (developers)performance.rst, line 407
```

Unknown directive type "prompt".

```
.. prompt:: bash $

# open my_file.py.callgrin with kcachegrind
kcachegrind my_file.py.prof
```

### Note

`yep` can be executed with the argument `--lines` or `-l` to compile a profiling report 'line by line'.

## Multi-core parallelism using `joblib.Parallel`

See [joblib documentation](#)

## A simple algorithmic trick: warm restarts

See the glossary entry for [warm\\_start](#)