CHAPTER

SEVEN

DEVELOPER'S GUIDE

7.1 Contributing

This project is a community effort, and everyone is welcome to contribute.

The project is hosted on https://github.com/scikit-learn/scikit-learn

The decision making process and governance structure of scikit-learn is laid out in the governance document: *Scikit-learn governance and decision-making*.

Scikit-learn is somewhat *selective* when it comes to adding new algorithms, and the best way to contribute and to help the project is to start working on known issues. See *Issues for New Contributors* to get started.

Our community, our values

We are a community based on openness and friendly, didactic, discussions.

We aspire to treat everybody equally, and value their contributions.

Decisions are made based on technical merit and consensus.

Code is not the only way to help the project. Reviewing pull requests, answering questions to help others on mailing lists or issues, organizing and teaching tutorials, working on the website, improving the documentation, are all priceless contributions.

We abide by the principles of openness, respect, and consideration of others of the Python Software Foundation: https://www.python.org/psf/codeofconduct/

In case you experience issues using this package, do not hesitate to submit a ticket to the GitHub issue tracker. You are also welcome to post feature requests or pull requests.

7.1.1 Ways to contribute

There are many ways to contribute to scikit-learn, with the most common ones being contribution of code or documentation to the project. Improving the documentation is no less important than improving the library itself. If you find a typo in the documentation, or have made improvements, do not hesitate to send an email to the mailing list or preferably submit a GitHub pull request. Full documentation can be found under the doc/ directory.

But there are many other ways to help. In particular answering queries on the issue tracker, investigating bugs, and *reviewing other developers' pull requests* are very valuable contributions that decrease the burden on the project maintainers.

Another way to contribute is to report issues you're facing, and give a "thumbs up" on issues that others reported and that are relevant to you. It also helps us if you spread the word: reference the project from your blog and articles, link to it from your website, or simply star to say "I use it":

In case a contribution/issue involves changes to the API principles or changes to dependencies or supported versions, it must be backed by a *Enhancement proposals (SLEPs)*, where a SLEP must be submitted as a pull-request to enhancement proposals using the SLEP template and follows the decision-making process outlined in *Scikit-learn governance and decision-making*.

Contributing to related projects

Scikit-learn thrives in an ecosystem of several related projects, which also may have relevant issues to work on, including smaller projects such as:

- scikit-learn-contrib
- joblib
- sphinx-gallery
- numpydoc
- liac-arff

and larger projects:

- numpy
- scipy
- matplotlib
- and so on.

Look for issues marked "help wanted" or similar. Helping these projects may help Scikit-learn too. See also *Related Projects*.

7.1.2 Submitting a bug report or a feature request

We use GitHub issues to track all bugs and feature requests; feel free to open an issue if you have found a bug or wish to see a feature implemented.

In case you experience issues using this package, do not hesitate to submit a ticket to the Bug Tracker. You are also welcome to post feature requests or pull requests.

It is recommended to check that your issue complies with the following rules before submitting:

- Verify that your issue is not being currently addressed by other issues or pull requests.
- If you are submitting an algorithm or feature request, please verify that the algorithm fulfills our new algorithm requirements.
- If you are submitting a bug report, we strongly encourage you to follow the guidelines in *How to make a good bug report*.

How to make a good bug report

When you submit an issue to Github, please do your best to follow these guidelines! This will make it a lot easier to provide you with good feedback:

- The ideal bug report contains a **short reproducible code snippet**, this way anyone can try to reproduce the bug easily (see this for more details). If your snippet is longer than around 50 lines, please link to a gist or a github repo.
- If not feasible to include a reproducible snippet, please be specific about what **estimators and/or functions are involved and the shape of the data**.
- If an exception is raised, please **provide the full traceback**.
- Please include your **operating system type and version number**, as well as your **Python, scikit-learn, numpy, and scipy versions**. This information can be found by running the following code snippet:

```
>>> import sklearn
>>> sklearn.show_versions()
```

Note: This utility function is only available in scikit-learn v0.20+. For previous versions, one has to explicitly run:

```
import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)
import sklearn; print("Scikit-Learn", sklearn.__version__)
```

- Please ensure all **code snippets and error messages are formatted in appropriate code blocks**. See Creating and highlighting code blocks for more details.

7.1.3 Contributing code

Note: To avoid duplicating work, it is highly advised that you search through the issue tracker and the PR list. If in doubt about duplicated work, or if you want to work on a non-trivial feature, it's recommended to first open an issue in the issue tracker to get some feedbacks from core developers.

How to contribute

The preferred way to contribute to scikit-learn is to fork the main repository on GitHub, then submit a "pull request" (PR):

1. Create an account on GitHub if you do not already have one.
2. Fork the project repository: click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub user account. For more details on how to fork a repository see this guide.
3. Clone your fork of the scikit-learn repo from your GitHub account to your local disk:

```
$ git clone git@github.com:YourLogin/scikit-learn.git
$ cd scikit-learn
```

4. Install the development dependencies:

```
$ pip install cython pytest flake8
```

5. Install scikit-learn in editable mode:

```
$ pip install --editable .
```

for more details about advanced installation, see the *Building from source* section.

6. Add the `upstream` remote. This saves a reference to the main scikit-learn repository, which you can use to keep your repository synchronized with the latest changes:

```
$ git remote add upstream https://github.com/scikit-learn/scikit-learn.git
```

7. Fetch the `upstream` and then create a branch to hold your development changes:

```
$ git fetch upstream
$ git checkout -b my-feature upstream/master
```

and start making changes. Always use a `feature` branch. It's good practice to never work on the `master` branch!

8. Develop the feature on your feature branch on your computer, using Git to do the version control. When you're done editing, add changed files using `git add` and then `git commit` files:

```
$ git add modified_files
$ git commit
```

to record your changes in Git, then push the changes to your GitHub account with:

```
$ git push -u origin my-feature
```

9. Follow these instructions to create a pull request from your fork. This will send an email to the committers. You may want to consider sending an email to the mailing list for more visibility.

Note: If you are modifying a Cython module, you have to re-run step 5 after modifications and before testing them.

It is often helpful to keep your local branch synchronized with the latest changes of the main scikit-learn repository:

```
$ git fetch upstream
$ git merge upstream/master
```

Subsequently, you might need to solve the conflicts. You can refer to the Git documentation related to resolving merge conflict using the command line.

Learning git:

The Git documentation and http://try.github.io are excellent resources to get started with git, and understanding all of the commands shown here.

Pull request checklist

Before a PR can be merged, it needs to be approved by two core developers. Please prefix the title of your pull request with `[MRG]` if the contribution is complete and should be subjected to a detailed review. An incomplete contribution – where you expect to do more work before receiving a full review – should be prefixed `[WIP]` (to indicate a work in progress) and changed to `[MRG]` when it matures. WIPs may be useful to: indicate you are working on something to avoid duplicated work, request broad review of functionality or API, or seek collaborators. WIPs often benefit from the inclusion of a task list in the PR description.

In order to ease the reviewing process, we recommend that your contribution complies with the following rules before marking a PR as `[MRG]`. The **bolded** ones are especially important:

1. **Give your pull request a helpful title** that summarises what your contribution does. This title will often become the commit message once merged so it should summarise your contribution for posterity. In some cases "Fix <ISSUE TITLE>" is enough. "Fix #<ISSUE NUMBER>" is never a good title.

2. **Make sure your code passes the tests**. The whole test suite can be run with `pytest`, but it is usually not recommended since it takes a long time. It is often enough to only run the test related to your changes: for example, if you changed something in `sklearn/linear_model/logistic.py`, running the following commands will usually be enough:

   - `pytest sklearn/linear_model/logistic.py` to make sure the doctest examples are correct
   - `pytest sklearn/linear_model/tests/test_logistic.py` to run the tests specific to the file
   - `pytest sklearn/linear_model` to test the whole *Generalized Linear Models* module
   - `pytest sklearn/doc/linear_model.rst` to make sure the user guide examples are correct.
   - `pytest sklearn/tests/test_common.py -k LogisticRegression` to run all our estimator checks (specifically for *`LogisticRegression`*, if that's the estimator you changed).

   There may be other failing tests, but they will be caught by the CI so you don't need to run the whole test suite locally. You can read more in *Testing and improving test coverage*.

3. **Make sure your code is properly commented and documented**, and **make sure the documentation renders properly**. To build the documentation, please refer to our *Documentation* guidelines. The CI will also build the docs: please refer to *Generated documentation on CircleCI*.

4. **Tests are necessary for enhancements to be accepted**. Bug-fixes or new features should be provided with non-regression tests. These tests verify the correct behavior of the fix or feature. In this manner, further modifications on the code base are granted to be consistent with the desired behavior. In the case of bug fixes, at the time of the PR, the non-regression tests should fail for the code base in the master branch and pass for the PR code.

5. **Make sure that your PR does not add PEP8 violations**. On a Unix-like system, you can run `make flake8-diff`. `flake8 path_to_file`, would work for any system, but please avoid reformatting parts of the file that your pull request doesn't change, as it distracts from code review.

6. Follow the *coding-guidelines* (see below).

7. When applicable, use the validation tools and scripts in the `sklearn.utils` submodule. A list of utility routines available for developers can be found in the *Utilities for Developers* page.

8. Often pull requests resolve one or more other issues (or pull requests). If merging your pull request means that some other issues/PRs should be closed, you should use keywords to create link to them (e.g., `Fixes #1234`; multiple issues/PRs are allowed as long as each one is preceded by a keyword). Upon merging, those issues/PRs will automatically be closed by GitHub. If your pull request is simply related to some other issues/PRs, create a link to them without using the keywords (e.g., `See also #1234`).

9. PRs should often substantiate the change, through benchmarks of performance and efficiency or through examples of usage. Examples also illustrate the features and intricacies of the library to users. Have a look at other examples in the examples/ directory for reference. Examples should demonstrate why the new functionality is useful in practice and, if possible, compare it to other methods available in scikit-learn.

10. New features often need to be illustrated with narrative documentation in the user guide, with small code snipets. If relevant, please also add references in the literature, with PDF links when possible.

11. The user guide should also include expected time and space complexity of the algorithm and scalability, e.g. "this algorithm can scale to a large number of samples > 100000, but does not scale in dimensionality: n_features is expected to be lower than 100".

You can also check our *Code Review Guidelines* to get an idea of what reviewers will expect.

You can check for common programming errors with the following tools:

- Code with a good unittest coverage (at least 80%, better 100%), check with:

```
$ pip install pytest pytest-cov
$ pytest --cov sklearn path/to/tests_for_package
```

  see also *Testing and improving test coverage*

Bonus points for contributions that include a performance analysis with a benchmark script and profiling output (please report on the mailing list or on the GitHub issue).

Also check out the *How to optimize for speed* guide for more details on profiling and Cython optimizations.

Note: The current state of the scikit-learn code base is not compliant with all of those guidelines, but we expect that enforcing those constraints on all new contributions will get the overall code base quality in the right direction.

Note: For two very well documented and more detailed guides on development workflow, please pay a visit to the Scipy Development Workflow - and the Astropy Workflow for Developers sections.

Continuous Integration (CI)

- Azure pipelines are used for testing scikit-learn on Linux, Mac and Windows, with different dependencies and settings.
- CircleCI is used to build the docs for viewing, for linting with flake8, and for testing with PyPy on Linux

Please note that if one of the following markers appear in the latest commit message, the following actions are taken.

Commit Message Marker	Action Taken by CI
[scipy-dev]	Add a Travis build with our dependencies (numpy, scipy, etc ...) development builds
[ci skip]	CI is skipped completely
[doc skip]	Docs are not built
[doc quick]	Docs built, but excludes example gallery plots
[doc build]	Docs built including example gallery plots

Stalled pull requests

As contributing a feature can be a lengthy process, some pull requests appear inactive but unfinished. In such a case, taking them over is a great service for the project.

A good etiquette to take over is:

- **Determine if a PR is stalled**
  - A pull request may have the label "stalled" or "help wanted" if we have already identified it as a candidate for other contributors.

  - To decide whether an inactive PR is stalled, ask the contributor if she/he plans to continue working on the PR in the near future. Failure to respond within 2 weeks with an activity that moves the PR forward suggests that the PR is stalled and will result in tagging that PR with "help wanted".

    Note that if a PR has received earlier comments on the contribution that have had no reply in a month, it is safe to assume that the PR is stalled and to shorten the wait time to one day.

    After a sprint, follow-up for un-merged PRs opened during sprint will be communicated to participants at the sprint, and those PRs will be tagged "sprint". PRs tagged with "sprint" can be reassigned or declared stalled by sprint leaders.

- **Taking over a stalled PR**: To take over a PR, it is important to comment on the stalled PR that you are taking over and to link from the new PR to the old one. The new PR should be created by pulling from the old one.

Issues for New Contributors

New contributors should look for the following tags when looking for issues. We strongly recommend that new contributors tackle "easy" issues first: this helps the contributor become familiar with the contribution workflow, and for the core devs to become acquainted with the contributor; besides which, we frequently underestimate how easy an issue is to solve!

good first issue tag

A great way to start contributing to scikit-learn is to pick an item from the list of good first issues in the issue tracker. Resolving these issues allow you to start contributing to the project without much prior knowledge. If you have already contributed to scikit-learn, you should look at Easy issues instead.

Easy tag

If you have already contributed to scikit-learn, another great way to contribute to scikit-learn is to pick an item from the list of Easy issues in the issue tracker. Your assistance in this area will be greatly appreciated by the more experienced developers as it helps free up their time to concentrate on other issues.

help wanted tag

We often use the help wanted tag to mark issues regardless of difficulty. Additionally, we use the help wanted tag to mark Pull Requests which have been abandoned by their original contributor and are available for someone to pick up where the original contributor left off. The list of issues with the help wanted tag can be found here .

Note that not all issues which need contributors will have this tag.

7.1.4 Documentation

We are glad to accept any sort of documentation: function docstrings, reStructuredText documents (like this one), tutorials, etc. reStructuredText documents live in the source code repository under the `doc/` directory.

You can edit the documentation using any text editor, and then generate the HTML output by typing `make` from the `doc/` directory. Alternatively, `make html` may be used to generate the documentation **with** the example gallery (which takes quite some time). The resulting HTML files will be placed in `_build/html/stable` and are viewable in a web browser.

Building the documentation

First, make sure you have *properly installed* the development version.

Building the documentation requires installing some additional packages:

```
pip install sphinx sphinx-gallery numpydoc matplotlib Pillow pandas scikit-image
```

To build the documentation, you need to be in the `doc` folder:

```
cd doc
```

In the vast majority of cases, you only need to generate the full web site, without the example gallery:

```
make
```

The documentation will be generated in the `_build/html/stable` directory. To also generate the example gallery you can use:

```
make html
```

This will run all the examples, which takes a while. If you only want to generate a few examples, you can use:

```
EXAMPLES_PATTERN=your_regex_goes_here make html
```

This is particularly useful if you are modifying a few examples.

Set the environment variable `NO_MATHJAX=1` if you intend to view the documentation in an offline setting.

To build the PDF manual, run:

```
make latexpdf
```

Warning: Sphinx version

While we do our best to have the documentation build under as many versions of Sphinx as possible, the different versions tend to behave slightly differently. To get the best results, you should use the same version as the one we used on CircleCI. Look at this github search to know the exact version.

Guidelines for writing documentation

It is important to keep a good compromise between mathematical and algorithmic details, and give intuition to the reader on what the algorithm does.

Basically, to elaborate on the above, it is best to always start with a small paragraph with a hand-waving explanation of what the method does to the data. Then, it is very helpful to point out why the feature is useful and when it should be used - the latter also including "big O" ($O\left(g\left(n\right)\right)$) complexities of the algorithm, as opposed to just *rules of thumb*, as the latter can be very machine-dependent. If those complexities are not available, then rules of thumb may be provided instead.

Secondly, a generated figure from an example (as mentioned in the previous paragraph) should then be included to further provide some intuition.

Next, one or two small code examples to show its use can be added.

Next, any math and equations, followed by references, can be added to further the documentation. Not starting the documentation with the maths makes it more friendly towards users that are just interested in what the feature will do, as opposed to how it works "under the hood".

Finally, follow the formatting rules below to make it consistently good:

- Add "See also" in docstrings for related classes/functions.
- "See also" in docstrings should be one line per reference, with a colon and an explanation, for example:

```
See also
--------
SelectKBest : Select features based on the k highest scores.
SelectFpr : Select features based on a false positive rate test.
```

- For unwritten formatting rules, try to follow existing good works:
  - For "References" in docstrings, see the Silhouette Coefficient (*sklearn.metrics.silhouette_score*).
- When editing reStructuredText (`.rst`) files, try to keep line length under 80 characters when possible (exceptions include links and tables).

Generated documentation on CircleCI

When you change the documentation in a pull request, CircleCI automatically builds it. To view the documentation generated by CircleCI:

- navigate to the bottom of your pull request page to see the CI statuses. You may need to click on "Show all checks" to see all the CI statuses.
- click on the CircleCI status with "doc" in the title.
- add `#artifacts` at the end of the URL. Note: you need to wait for the CircleCI build to finish before being able to look at the artifacts.
- once the artifacts are visible, navigate to `doc/_changed.html` to see a list of documentation pages that are likely to be affected by your pull request. Navigate to `doc/index.html` to see the full generated html documentation.

If you often need to look at the documentation generated by CircleCI, e.g. when reviewing pull requests, you may find *this tip* very handy.

7.1.5 Testing and improving test coverage

High-quality unit testing is a corner-stone of the scikit-learn development process. For this purpose, we use the pytest package. The tests are functions appropriately named, located in `tests` subdirectories, that check the validity of the algorithms and the different options of the code.

The full scikit-learn tests can be run using 'make' in the root folder. Alternatively, running 'pytest' in a folder will run all the tests of the corresponding subpackages.

We expect code coverage of new features to be at least around 90%.

For guidelines on how to use `pytest` efficiently, see the *Useful pytest aliases and flags*.

Writing matplotlib related tests

Test fixtures ensure that a set of tests will be executing with the appropriate initialization and cleanup. The scikit-learn test suite implements a fixture which can be used with `matplotlib`.

`pyplot` The `pyplot` fixture should be used when a test function is dealing with `matplotlib`. `matplotlib` is a soft dependency and is not required. This fixture is in charge of skipping the tests if `matplotlib` is not installed. In addition, figures created during the tests will be automatically closed once the test function has been executed.

To use this fixture in a test function, one needs to pass it as an argument:

```
def test_requiring_mpl_fixture(pyplot):
    # you can now safely use matplotlib
```

Workflow to improve test coverage

To test code coverage, you need to install the coverage package in addition to pytest.

1. **Run 'make test-coverage'. The output lists for each file the line** numbers that are not tested.
2. **Find a low hanging fruit, looking at which lines are not tested,** write or adapt a test specifically for these lines.
3. Loop.

Issue Tracker Tags

All issues and pull requests on the GitHub issue tracker should have (at least) one of the following tags:

Bug / Crash Something is happening that clearly shouldn't happen. Wrong results as well as unexpected errors from estimators go here.

Cleanup / Enhancement Improving performance, usability, consistency.

Documentation Missing, incorrect or sub-standard documentations and examples.

New Feature Feature requests and pull requests implementing a new feature.

There are four other tags to help new contributors:

good first issue This issue is ideal for a first contribution to scikit-learn. Ask for help if the formulation is unclear. If you have already contributed to scikit-learn, look at Easy issues instead.

Easy This issue can be tackled without much prior experience.

Moderate Might need some knowledge of machine learning or the package, but is still approachable for someone new to the project.

help wanted This tag marks an issue which currently lacks a contributor or a PR that needs another contributor to take over the work. These issues can range in difficulty, and may not be approachable for new contributors. Note that not all issues which need contributors will have this tag.

7.1.6 Coding guidelines

The following are some guidelines on how new code should be written. Of course, there are special cases and there will be exceptions to these rules. However, following these rules when submitting new code makes the review easier so new code can be integrated in less time.

Uniformly formatted code makes it easier to share code ownership. The scikit-learn project tries to closely follow the official Python guidelines detailed in PEP8 that detail how code should be formatted and indented. Please read it and follow it.

In addition, we add the following guidelines:

- Use underscores to separate words in non class names: `n_samples` rather than `nsamples`.
- Avoid multiple statements on one line. Prefer a line return after a control flow statement (`if`/`for`).
- Use relative imports for references inside scikit-learn.
- Unit tests are an exception to the previous rule; they should use absolute imports, exactly as client code would. A corollary is that, if `sklearn.foo` exports a class or function that is implemented in `sklearn.foo.bar.baz`, the test should import it from `sklearn.foo`.
- **Please don't use** `import *` **in any case**. It is considered harmful by the official Python recommendations. It makes the code harder to read as the origin of symbols is no longer explicitly referenced, but most important, it prevents using a static analysis tool like pyflakes to automatically find bugs in scikit-learn.
- Use the numpy docstring standard in all your docstrings.

A good example of code that we like can be found here.

Input validation

The module *`sklearn.utils`* contains various functions for doing input validation and conversion. Sometimes, `np.asarray` suffices for validation; do *not* use `np.asanyarray` or `np.atleast_2d`, since those let NumPy's `np.matrix` through, which has a different API (e.g., `*` means dot product on `np.matrix`, but Hadamard product on `np.ndarray`).

In other cases, be sure to call *`check_array`* on any array-like argument passed to a scikit-learn API function. The exact parameters to use depends mainly on whether and which `scipy.sparse` matrices must be accepted.

For more information, refer to the *Utilities for Developers* page.

Random Numbers

If your code depends on a random number generator, do not use `numpy.random.random()` or similar routines. To ensure repeatability in error checking, the routine should accept a keyword `random_state` and use this to construct a `numpy.random.RandomState` object. See *`sklearn.utils.check_random_state`* in *Utilities for Developers*.

Here's a simple example of code using some of the above guidelines:

```
from sklearn.utils import check_array, check_random_state

def choose_random_sample(X, random_state=0):
    """
    Choose a random point from X

    Parameters
    ----------
    X : array-like, shape (n_samples, n_features)
        array representing the data
    random_state : RandomState or an int seed (0 by default)
        A random number generator instance to define the state of the
        random permutations generator.
```

```
    Returns
    -------
    x : numpy array, shape (n_features,)
        A random point selected from X
    """
    X = check_array(X)
    random_state = check_random_state(random_state)
    i = random_state.randint(X.shape[0])
    return X[i]
```

If you use randomness in an estimator instead of a freestanding function, some additional guidelines apply.

First off, the estimator should take a `random_state` argument to its `__init__` with a default value of `None`. It should store that argument's value, **unmodified**, in an attribute `random_state`. `fit` can call `check_random_state` on that attribute to get an actual random number generator. If, for some reason, randomness is needed after `fit`, the RNG should be stored in an attribute `random_state_`. The following example should make this clear:

```
class GaussianNoise(BaseEstimator, TransformerMixin):
    """This estimator ignores its input and returns random Gaussian noise.

    It also does not adhere to all scikit-learn conventions,
    but showcases how to handle randomness.
    """

    def __init__(self, n_components=100, random_state=None):
        self.random_state = random_state

    # the arguments are ignored anyway, so we make them optional
    def fit(self, X=None, y=None):
        self.random_state_ = check_random_state(self.random_state)

    def transform(self, X):
        n_samples = X.shape[0]
        return self.random_state_.randn(n_samples, n_components)
```

The reason for this setup is reproducibility: when an estimator is `fit` twice to the same data, it should produce an identical model both times, hence the validation in `fit`, not `__init__`.

Deprecation

If any publicly accessible method, function, attribute or parameter is renamed, we still support the old one for two releases and issue a deprecation warning when it is called/passed/accessed. E.g., if the function `zero_one` is renamed to `zero_one_loss`, we add the decorator `deprecated` (from `sklearn.utils`) to `zero_one` and call `zero_one_loss` from that function:

```
from ..utils import deprecated

def zero_one_loss(y_true, y_pred, normalize=True):
    # actual implementation
    pass

@deprecated("Function 'zero_one' was renamed to 'zero_one_loss' "
            "in version 0.13 and will be removed in release 0.15. "
            "Default behavior is changed from 'normalize=False' to "
            "'normalize=True'")
```

```
def zero_one(y_true, y_pred, normalize=False):
    return zero_one_loss(y_true, y_pred, normalize)
```

If an attribute is to be deprecated, use the decorator `deprecated` on a property. Please note that the `property` decorator should be placed before the `deprecated` decorator for the docstrings to be rendered properly. E.g., renaming an attribute `labels_` to `classes_` can be done as:

```
@deprecated("Attribute labels_ was deprecated in version 0.13 and "
            "will be removed in 0.15. Use 'classes_' instead")
@property
def labels_(self):
    return self.classes_
```

If a parameter has to be deprecated, use `DeprecationWarning` appropriately. In the following example, k is deprecated and renamed to n_clusters:

```
import warnings

def example_function(n_clusters=8, k='not_used'):
    if k != 'not_used':
        warnings.warn("'k' was renamed to n_clusters in version 0.13 and "
                      "will be removed in 0.15.", DeprecationWarning)
        n_clusters = k
```

When the change is in a class, we validate and raise warning in `fit`:

```
import warnings

class ExampleEstimator(BaseEstimator):
    def __init__(self, n_clusters=8, k='not_used'):
        self.n_clusters = n_clusters
        self.k = k

    def fit(self, X, y):
        if self.k != 'not_used':
            warnings.warn("'k' was renamed to n_clusters in version 0.13 and "
                          "will be removed in 0.15.", DeprecationWarning)
            self._n_clusters = self.k
        else:
            self._n_clusters = self.n_clusters
```

As in these examples, the warning message should always give both the version in which the deprecation happened and the version in which the old behavior will be removed. If the deprecation happened in version 0.x-dev, the message should say deprecation occurred in version 0.x and the removal will be in 0.(x+2), so that users will have enough time to adapt their code to the new behaviour. For example, if the deprecation happened in version 0.18-dev, the message should say it happened in version 0.18 and the old behavior will be removed in version 0.20.

In addition, a deprecation note should be added in the docstring, recalling the same information as the deprecation warning as explained above. Use the `.. deprecated::` directive:

```
.. deprecated:: 0.13
   ``k`` was renamed to ``n_clusters`` in version 0.13 and will be removed
   in 0.15.
```

What's more, a deprecation requires a test which ensures that the warning is raised in relevant cases but not in other cases. The warning should be caught in all other tests (using e.g., `@pytest.mark.filterwarnings`), and there should be no warning in the examples.

Change the default value of a parameter

If the default value of a parameter needs to be changed, please replace the default value with a specific value (e.g., `warn`) and raise `FutureWarning` when users are using the default value. In the following example, we change the default value of `n_clusters` from 5 to 10 (current version is 0.20):

```
import warnings

def example_function(n_clusters='warn'):
    if n_clusters == 'warn':
        warnings.warn("The default value of n_clusters will change from "
                      "5 to 10 in 0.22.", FutureWarning)
        n_clusters = 5
```

When the change is in a class, we validate and raise warning in `fit`:

```
import warnings

class ExampleEstimator:
    def __init__(self, n_clusters='warn'):
        self.n_clusters = n_clusters

    def fit(self, X, y):
        if self.n_clusters == 'warn':
          warnings.warn("The default value of n_clusters will change from "
                        "5 to 10 in 0.22.", FutureWarning)
          self._n_clusters = 5
```

Similar to deprecations, the warning message should always give both the version in which the change happened and the version in which the old behavior will be removed. The docstring needs to be updated accordingly. We need a test which ensures that the warning is raised in relevant cases but not in other cases. The warning should be caught in all other tests (using e.g., `@pytest.mark.filterwarnings`), and there should be no warning in the examples.

Python versions supported

Since scikit-learn 0.21, only Python 3.5 and newer is supported.

7.1.7 Code Review Guidelines

Reviewing code contributed to the project as PRs is a crucial component of scikit-learn development. We encourage anyone to start reviewing code of other developers. The code review process is often highly educational for everybody involved. This is particularly appropriate if it is a feature you would like to use, and so can respond critically about whether the PR meets your needs. While each pull request needs to be signed off by two core developers, you can speed up this process by providing your feedback.

Here are a few important aspects that need to be covered in any code review, from high-level questions to a more detailed check-list.

- Do we want this in the library? Is it likely to be used? Do you, as a scikit-learn user, like the change and intend to use it? Is it in the scope of scikit-learn? Will the cost of maintaining a new feature be worth its benefits?
- Is the code consistent with the API of scikit-learn? Are public functions/classes/parameters well named and intuitively designed?
- Are all public functions/classes and their parameters, return types, and stored attributes named according to scikit-learn conventions and documented clearly?

- Is any new functionality described in the user-guide and illustrated with examples?
- Is every public function/class tested? Are a reasonable set of parameters, their values, value types, and combinations tested? Do the tests validate that the code is correct, i.e. doing what the documentation says it does? If the change is a bug-fix, is a non-regression test included? Look at this to get started with testing in Python.
- Do the tests pass in the continuous integration build? If appropriate, help the contributor understand why tests failed.
- Do the tests cover every line of code (see the coverage report in the build log)? If not, are the lines missing coverage good exceptions?
- Is the code easy to read and low on redundancy? Should variable names be improved for clarity or consistency? Should comments be added? Should comments be removed as unhelpful or extraneous?
- Could the code easily be rewritten to run much more efficiently for relevant settings?
- Is the code backwards compatible with previous versions? (or is a deprecation cycle necessary?)
- Will the new code add any dependencies on other libraries? (this is unlikely to be accepted)
- Does the documentation render properly (see the *Documentation* section for more details), and are the plots instructive?

Standard replies for reviewing includes some frequent comments that reviewers may make.

7.1.8 APIs of scikit-learn objects

To have a uniform API, we try to have a common basic API for all the objects. In addition, to avoid the proliferation of framework code, we try to adopt simple conventions and limit to a minimum the number of methods an object must implement.

Elements of the scikit-learn API are described more definitively in the *Glossary of Common Terms and API Elements*.

Different objects

The main objects in scikit-learn are (one class can implement multiple interfaces):

Estimator The base object, implements a `fit` method to learn from data, either:

```
estimator = estimator.fit(data, targets)
```

or:

```
estimator = estimator.fit(data)
```

Predictor For supervised learning, or some unsupervised problems, implements:

```
prediction = predictor.predict(data)
```

Classification algorithms usually also offer a way to quantify certainty of a prediction, either using `decision_function` or `predict_proba`:

```
probability = predictor.predict_proba(data)
```

Transformer For filtering or modifying the data, in a supervised or unsupervised way, implements:

```
new_data = transformer.transform(data)
```

When fitting and transforming can be performed much more efficiently together than separately, implements:

```
new_data = transformer.fit_transform(data)
```

Model A model that can give a goodness of fit measure or a likelihood of unseen data, implements (higher is better):

```
score = model.score(data)
```

Estimators

The API has one predominant object: the estimator. A estimator is an object that fits a model based on some training data and is capable of inferring some properties on new data. It can be, for instance, a classifier or a regressor. All estimators implement the fit method:

```
estimator.fit(X, y)
```

All built-in estimators also have a `set_params` method, which sets data-independent parameters (overriding previous parameter values passed to `__init__`).

All estimators in the main scikit-learn codebase should inherit from `sklearn.base.BaseEstimator`.

Instantiation

This concerns the creation of an object. The object's `__init__` method might accept constants as arguments that determine the estimator's behavior (like the C constant in SVMs). It should not, however, take the actual training data as an argument, as this is left to the `fit()` method:

```
clf2 = SVC(C=2.3)
clf3 = SVC([[1, 2], [2, 3]], [-1, 1]) # WRONG!
```

The arguments accepted by `__init__` should all be keyword arguments with a default value. In other words, a user should be able to instantiate an estimator without passing any arguments to it. The arguments should all correspond to hyperparameters describing the model or the optimisation problem the estimator tries to solve. These initial arguments (or parameters) are always remembered by the estimator. Also note that they should not be documented under the "Attributes" section, but rather under the "Parameters" section for that estimator.

In addition, **every keyword argument accepted by `__init__` should correspond to an attribute on the instance**. Scikit-learn relies on this to find the relevant attributes to set on an estimator when doing model selection.

To summarize, an `__init__` should look like:

```
def __init__(self, param1=1, param2=2):
    self.param1 = param1
    self.param2 = param2
```

There should be no logic, not even input validation, and the parameters should not be changed. The corresponding logic should be put where the parameters are used, typically in `fit`. The following is wrong:

```
def __init__(self, param1=1, param2=2, param3=3):
    # WRONG: parameters should not be modified
    if param1 > 1:
        param2 += 1
    self.param1 = param1
```

```
    # WRONG: the object's attributes should have exactly the name of
    # the argument in the constructor
    self.param3 = param2
```

The reason for postponing the validation is that the same validation would have to be performed in `set_params`, which is used in algorithms like `GridSearchCV`.

Fitting

The next thing you will probably want to do is to estimate some parameters in the model. This is implemented in the `fit()` method.

The `fit()` method takes the training data as arguments, which can be one array in the case of unsupervised learning, or two arrays in the case of supervised learning.

Note that the model is fitted using X and y, but the object holds no reference to X and y. There are, however, some exceptions to this, as in the case of precomputed kernels where this data must be stored for use by the predict method.

Parameters	
X	array-like, shape (n_samples, n_features)
y	array, shape (n_samples,)
kwargs	optional data-dependent parameters.

`X.shape[0]` should be the same as `y.shape[0]`. If this requisite is not met, an exception of type `ValueError` should be raised.

`y` might be ignored in the case of unsupervised learning. However, to make it possible to use the estimator as part of a pipeline that can mix both supervised and unsupervised transformers, even unsupervised estimators need to accept a `y=None` keyword argument in the second position that is just ignored by the estimator. For the same reason, `fit_predict`, `fit_transform`, `score` and `partial_fit` methods need to accept a `y` argument in the second place if they are implemented.

The method should return the object (`self`). This pattern is useful to be able to implement quick one liners in an IPython session such as:

```
y_predicted = SVC(C=100).fit(X_train, y_train).predict(X_test)
```

Depending on the nature of the algorithm, `fit` can sometimes also accept additional keywords arguments. However, any parameter that can have a value assigned prior to having access to the data should be an `__init__` keyword argument. **fit parameters should be restricted to directly data dependent variables**. For instance a Gram matrix or an affinity matrix which are precomputed from the data matrix `X` are data dependent. A tolerance stopping criterion `tol` is not directly data dependent (although the optimal value according to some scoring function probably is).

When `fit` is called, any previous call to `fit` should be ignored. In general, calling `estimator.fit(X1)` and then `estimator.fit(X2)` should be the same as only calling `estimator.fit(X2)`. However, this may not be true in practice when `fit` depends on some random process, see *random_state*. Another exception to this rule is when the hyper-parameter `warm_start` is set to `True` for estimators that support it. `warm_start=True` means that the previous state of the trainable parameters of the estimator are reused instead of using the default initialization strategy.

Estimated Attributes

Attributes that have been estimated from the data must always have a name ending with trailing underscore, for example the coefficients of some regression estimator would be stored in a `coef_` attribute after `fit` has been called.

The estimated attributes are expected to be overridden when you call `fit` a second time.

Optional Arguments

In iterative algorithms, the number of iterations should be specified by an integer called `n_iter`.

Pairwise Attributes

An estimator that accept `X` of shape `(n_samples, n_samples)` and defines a *_pairwise* property equal to `True` allows for cross-validation of the dataset, e.g. when `X` is a precomputed kernel matrix. Specifically, the *_pairwise* property is used by `utils.metaestimators._safe_split` to slice rows and columns.

7.1.9 Rolling your own estimator

If you want to implement a new estimator that is scikit-learn-compatible, whether it is just for you or for contributing it to scikit-learn, there are several internals of scikit-learn that you should be aware of in addition to the scikit-learn API outlined above. You can check whether your estimator adheres to the scikit-learn interface and standards by running *utils.estimator_checks.check_estimator* on the class:

```
>>> from sklearn.utils.estimator_checks import check_estimator
>>> from sklearn.svm import LinearSVC
>>> check_estimator(LinearSVC)  # passes
```

The main motivation to make a class compatible to the scikit-learn estimator interface might be that you want to use it together with model evaluation and selection tools such as *model_selection.GridSearchCV* and *pipeline.Pipeline*.

Before detailing the required interface below, we describe two ways to achieve the correct interface more easily.

Project template:

We provide a project template which helps in the creation of Python packages containing scikit-learn compatible estimators. It provides:

- an initial git repository with Python package directory structure
- a template of a scikit-learn estimator
- an initial test suite including use of `check_estimator`
- directory structures and scripts to compile documentation and example galleries
- scripts to manage continuous integration (testing on Linux and Windows)
- instructions from getting started to publishing on PyPi

`BaseEstimator` and mixins:

We tend to use "duck typing", so building an estimator which follows the API suffices for compatibility, without needing to inherit from or even import any scikit-learn classes.

However, if a dependency on scikit-learn is acceptable in your code, you can prevent a lot of boilerplate code by deriving a class from `BaseEstimator` and optionally the mixin classes in `sklearn.base`. For example, below is a custom classifier, with more examples included in the scikit-learn-contrib project template.

```
>>> import numpy as np
>>> from sklearn.base import BaseEstimator, ClassifierMixin
>>> from sklearn.utils.validation import check_X_y, check_array, check_is_fitted
>>> from sklearn.utils.multiclass import unique_labels
>>> from sklearn.metrics import euclidean_distances
>>> class TemplateClassifier(BaseEstimator, ClassifierMixin):
...
...     def __init__(self, demo_param='demo'):
...         self.demo_param = demo_param
...
...     def fit(self, X, y):
...
...         # Check that X and y have correct shape
...         X, y = check_X_y(X, y)
...         # Store the classes seen during fit
...         self.classes_ = unique_labels(y)
...
...         self.X_ = X
...         self.y_ = y
...         # Return the classifier
...         return self
...
...     def predict(self, X):
...
...         # Check is fit had been called
...         check_is_fitted(self, ['X_', 'y_'])
...
...         # Input validation
...         X = check_array(X)
...
...         closest = np.argmin(euclidean_distances(X, self.X_), axis=1)
...         return self.y_[closest]
```

get_params and set_params

All scikit-learn estimators have `get_params` and `set_params` functions. The `get_params` function takes no arguments and returns a dict of the `__init__` parameters of the estimator, together with their values. It must take one keyword argument, `deep`, which receives a boolean value that determines whether the method should return the parameters of sub-estimators (for most estimators, this can be ignored). The default value for `deep` should be true.

The `set_params` on the other hand takes as input a dict of the form `'parameter': value` and sets the parameter of the estimator using this dict. Return value must be estimator itself.

While the `get_params` mechanism is not essential (see *Cloning* below), the `set_params` function is necessary as it is used to set parameters during grid searches.

The easiest way to implement these functions, and to get a sensible `__repr__` method, is to inherit from `sklearn.base.BaseEstimator`. If you do not want to make your code dependent on scikit-learn, the easiest way to implement the interface is:

```
def get_params(self, deep=True):
    # suppose this estimator has parameters "alpha" and "recursive"
    return {"alpha": self.alpha, "recursive": self.recursive}
```

```
def set_params(self, **parameters):
    for parameter, value in parameters.items():
        setattr(self, parameter, value)
    return self
```

Parameters and init

As *model_selection.GridSearchCV* uses set_params to apply parameter setting to estimators, it is essential that calling set_params has the same effect as setting parameters using the __init__ method. The easiest and recommended way to accomplish this is to **not do any parameter validation in** __init__. All logic behind estimator parameters, like translating string arguments into functions, should be done in fit.

Also it is expected that parameters with trailing _ are **not to be set inside the** __init__ **method**. All and only the public attributes set by fit have a trailing _. As a result the existence of parameters with trailing _ is used to check if the estimator has been fitted.

Cloning

For use with the model_selection module, an estimator must support the base.clone function to replicate an estimator. This can be done by providing a get_params method. If get_params is present, then clone(estimator) will be an instance of type(estimator) on which set_params has been called with clones of the result of estimator.get_params().

Objects that do not provide this method will be deep-copied (using the Python standard function copy.deepcopy) if safe=False is passed to clone.

Pipeline compatibility

For an estimator to be usable together with pipeline.Pipeline in any but the last step, it needs to provide a fit or fit_transform function. To be able to evaluate the pipeline on any data but the training set, it also needs to provide a transform function. There are no special requirements for the last step in a pipeline, except that it has a fit function. All fit and fit_transform functions must take arguments X, y, even if y is not used. Similarly, for score to be usable, the last step of the pipeline needs to have a score function that accepts an optional y.

Estimator types

Some common functionality depends on the kind of estimator passed. For example, cross-validation in *model_selection.GridSearchCV* and *model_selection.cross_val_score* defaults to being stratified when used on a classifier, but not otherwise. Similarly, scorers for average precision that take a continuous prediction need to call decision_function for classifiers, but predict for regressors. This distinction between classifiers and regressors is implemented using the _estimator_type attribute, which takes a string value. It should be "classifier" for classifiers and "regressor" for regressors and "clusterer" for clustering methods, to work as expected. Inheriting from ClassifierMixin, RegressorMixin or ClusterMixin will set the attribute automatically. When a meta-estimator needs to distinguish among estimator types, instead of checking _estimator_type directly, helpers like *base.is_classifier* should be used.

Specific models

Classifiers should accept y (target) arguments to fit that are sequences (lists, arrays) of either strings or integers. They should not assume that the class labels are a contiguous range of integers; instead, they should store a list of

classes in a `classes_` attribute or property. The order of class labels in this attribute should match the order in which `predict_proba`, `predict_log_proba` and `decision_function` return their values. The easiest way to achieve this is to put:

```
self.classes_, y = np.unique(y, return_inverse=True)
```

in `fit`. This returns a new `y` that contains class indexes, rather than labels, in the range [0, `n_classes`).

A classifier's `predict` method should return arrays containing class labels from `classes_`. In a classifier that implements `decision_function`, this can be achieved with:

```
def predict(self, X):
    D = self.decision_function(X)
    return self.classes_[np.argmax(D, axis=1)]
```

In linear models, coefficients are stored in an array called `coef_`, and the independent term is stored in `intercept_`. `sklearn.linear_model.base` contains a few base classes and mixins that implement common linear model patterns.

The `sklearn.utils.multiclass` module contains useful functions for working with multiclass and multilabel problems.

Estimator Tags

Warning: The estimator tags are experimental and the API is subject to change.

Scikit-learn introduced estimator tags in version 0.21. These are annotations of estimators that allow programmatic inspection of their capabilities, such as sparse matrix support, supported output types and supported methods. The estimator tags are a dictionary returned by the method `_get_tags()`. These tags are used by the common tests and the *`sklearn.utils.estimator_checks.check_estimator`* function to decide what tests to run and what input data is appropriate. Tags can depend on estimator parameters or even system architecture and can in general only be determined at runtime.

The default value of all tags except for `X_types` is `False`. These are defined in the `BaseEstimator` class.

The current set of estimator tags are:

non_deterministic whether the estimator is not deterministic given a fixed `random_state`

requires_positive_data - unused for now whether the estimator requires positive X.

no_validation whether the estimator skips input-validation. This is only meant for stateless and dummy transformers!

multioutput - unused for now whether a regressor supports multi-target outputs or a classifier supports multi-class multi-output.

multilabel whether the estimator supports multilabel output

stateless whether the estimator needs access to data for fitting. Even though an estimator is stateless, it might still need a call to `fit` for initialization.

allow_nan whether the estimator supports data with missing values encoded as np.NaN

poor_score whether the estimator fails to provide a "reasonable" test-set score, which currently for regression is an R2 of 0.5 on a subset of the boston housing dataset, and for classification an accuracy of 0.83 on `make_blobs(n_samples=300, random_state=0)`. These datasets and values are based on current estimators in sklearn and might be replaced by something more systematic.

multioutput_only whether estimator supports only multi-output classification or regression.

_skip_test whether to skip common tests entirely. Don't use this unless you have a *very good* reason.

X_types Supported input types for X as list of strings. Tests are currently only run if '2darray' is contained in the list, signifying that the estimator takes continuous 2d numpy arrays as input. The default value is ['2darray']. Other possible types are `'string'`, `'sparse'`, `'categorical'`, `dict`, `'1dlabels'` and `'2dlabels'`. The goal is that in the future the supported input type will determine the data used during testing, in particular for `'string'`, `'sparse'` and `'categorical'` data. For now, the test for sparse data do not make use of the `'sparse'` tag.

To override the tags of a child class, one must define the `_more_tags()` method and return a dict with the desired tags, e.g:

```
class MyMultiOutputEstimator(BaseEstimator):

    def _more_tags(self):
        return {'multioutput_only': True,
                'non_deterministic': True}
```

In addition to the tags, estimators also need to declare any non-optional parameters to `__init__` in the `_required_parameters` class attribute, which is a list or tuple. If `_required_parameters` is only `["estimator"]` or `["base_estimator"]`, then the estimator will be instantiated with an instance of `LinearDiscriminantAnalysis` (or `RidgeRegression` if the estimator is a regressor) in the tests. The choice of these two models is somewhat idiosyncratic but both should provide robust closed-form solutions.

7.1.10 Reading the existing code base

Reading and digesting an existing code base is always a difficult exercise that takes time and experience to master. Even though we try to write simple code in general, understanding the code can seem overwhelming at first, given the sheer size of the project. Here is a list of tips that may help make this task easier and faster (in no particular order).

- Get acquainted with the *APIs of scikit-learn objects*: understand what *fit*, *predict*, *transform*, etc. are used for.
- Before diving into reading the code of a function / class, go through the docstrings first and try to get an idea of what each parameter / attribute is doing. It may also help to stop a minute and think *how would I do this myself if I had to?*
- The trickiest thing is often to identify which portions of the code are relevant, and which are not. In scikit-learn **a lot** of input checking is performed, especially at the beginning of the *fit* methods. Sometimes, only a very small portion of the code is doing the actual job. For example looking at the `fit()` method of *`sklearn.linear_model.LinearRegression`*, what you're looking for might just be the call the `scipy.linalg.lstsq`, but it is buried into multiple lines of input checking and the handling of different kinds of parameters.
- Due to the use of Inheritance, some methods may be implemented in parent classes. All estimators inherit at least from *`BaseEstimator`*, and from a `Mixin` class (e.g. *`ClassifierMixin`*) that enables default behaviour depending on the nature of the estimator (classifier, regressor, transformer, etc.).
- Sometimes, reading the tests for a given function will give you an idea of what its intended purpose is. You can use `git grep` (see below) to find all the tests written for a function. Most tests for a specific function/class are placed under the `tests/` folder of the module
- You'll often see code looking like this: `out = Parallel(...)(delayed(some_function)(param) for param in some_iterable)`. This runs `some_function` in parallel using Joblib. `out` is then an iterable containing the values returned by `some_function` for each call.
- We use Cython to write fast code. Cython code is located in `.pyx` and `.pxd` files. Cython code has a more C-like flavor: we use pointers, perform manual memory allocation, etc. Having some minimal experience in C

/ C++ is pretty much mandatory here.

- Master your tools.
  - With such a big project, being efficient with your favorite editor or IDE goes a long way towards digesting the code base. Being able to quickly jump (or *peek*) to a function/class/attribute definition helps a lot. So does being able to quickly see where a given name is used in a file.
  - git also has some built-in killer features. It is often useful to understand how a file changed over time, using e.g. `git blame` (manual). This can also be done directly on GitHub. `git grep` (examples) is also extremely useful to see every occurrence of a pattern (e.g. a function call or a variable) in the code base.

7.2 Developers' Tips and Tricks

7.2.1 Productivity and sanity-preserving tips

In this section we gather some useful advice and tools that may increase your quality-of-life when reviewing pull requests, running unit tests, and so forth. Some of these tricks consist of userscripts that require a browser extension such as TamperMonkey or GreaseMonkey; to set up userscripts you must have one of these extensions installed, enabled and running. We provide userscripts as GitHub gists; to install them, click on the "Raw" button on the gist page.

Viewing the rendered HTML documentation for a pull request

We use CircleCI to build the HTML documentation for every pull request. To access that documentation, instructions are provided in the *documentation section of the contributor guide*. To save you a few clicks, we provide a userscript that adds a button to every PR. After installing the userscript, navigate to any GitHub PR; a new button labeled "See CircleCI doc for this PR" should appear in the top-right area.

Folding and unfolding outdated diffs on pull requests

GitHub hides discussions on PRs when the corresponding lines of code have been changed in the mean while. This userscript provides a shortcut (Control-Alt-P at the time of writing but look at the code to be sure) to unfold all such hidden discussions at once, so you can catch up.

Checking out pull requests as remote-tracking branches

In your local fork, add to your `.git/config`, under the `[remote "upstream"]` heading, the line:

```
fetch = +refs/pull/*/head:refs/remotes/upstream/pr/*
```

You may then use `git checkout pr/PR_NUMBER` to navigate to the code of the pull-request with the given number. (Read more in this gist.)

Display code coverage in pull requests

To overlay the code coverage reports generated by the CodeCov continuous integration, consider this browser extension. The coverage of each line will be displayed as a color background behind the line number.

Useful pytest aliases and flags

The full test suite takes fairly long to run. For faster iterations, it is possibly to select a subset of tests using pytest selectors. In particular, one can run a single test based on its node ID:

```
pytest -v sklearn/linear_model/tests/test_logistic.py::test_sparsify
```

or use the -k pytest parameter to select tests based on their name. For instance,:

```
pytest sklearn/tests/test_common.py -v -k LogisticRegression
```

will run all *common tests* for the `LogisticRegression` estimator.

When a unit test fails, the following tricks can make debugging easier:

1. The command line argument `pytest -l` instructs pytest to print the local variables when a failure occurs.
2. The argument `pytest --pdb` drops into the Python debugger on failure. To instead drop into the rich IPython debugger `ipdb`, you may set up a shell alias to:

   ```
   pytest --pdbcls=IPython.terminal.debugger:TerminalPdb --capture no
   ```

Other `pytest` options that may become useful include:

- `-x` which exits on the first failed test
- `--lf` to rerun the tests that failed on the previous run
- `--ff` to rerun all previous tests, running the ones that failed first
- `-s` so that pytest does not capture the output of `print()` statements
- `--tb=short` or `--tb=line` to control the length of the logs

Since our continuous integration tests will error if `DeprecationWarning` or `FutureWarning` aren't properly caught, it is also recommended to run `pytest` along with the `-Werror::DeprecationWarning` and `-Werror::FutureWarning` flags.

Standard replies for reviewing

It may be helpful to store some of these in GitHub's saved replies for reviewing:

Issue: Usage questions

```
You're asking a usage question. The issue tracker is mainly for bugs and new␣
↪features. For usage questions, it is recommended to try [Stack Overflow](https:/
↪/stackoverflow.com/questions/tagged/scikit-learn) or [the Mailing List](https://
↪mail.python.org/mailman/listinfo/scikit-learn).
```

Issue: You're welcome to update the docs

```
Please feel free to offer a pull request updating the documentation if you feel␣
↪it could be improved.
```

Issue: Self-contained example for bug

```
Please provide [self-contained example code](https://stackoverflow.com/help/mcve),
↪ including imports and data (if possible), so that other contributors can just␣
↪run it and reproduce your issue. Ideally your example code should be minimal.
```

Issue: Software versions

```
To help diagnose your issue, please paste the output of:
```py
import sklearn; sklearn.show_versions()
```
Thanks.
```

Issue: Code blocks

```
Readability can be greatly improved if you [format](https://help.github.com/
↪articles/creating-and-highlighting-code-blocks/) your code snippets and␣
↪complete error messages appropriately. For example:

    ```python
    print(something)
    ```
generates:
```python
print(something)
```
And:

    ```pytb
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    ImportError: No module named 'hello'
    ```
generates:
```pytb
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'hello'
```
You can edit your issue descriptions and comments at any time to improve␣
↪readability. This helps maintainers a lot. Thanks!
```

Issue/Comment: Linking to code

```
Friendly advice: for clarity's sake, you can link to code like [this](https://
↪help.github.com/articles/creating-a-permanent-link-to-a-code-snippet/).
```

Issue/Comment: Linking to comments

```
Please use links to comments, which make it a lot easier to see what you are␣
↪referring to, rather than just linking to the issue. See [this](https://
↪stackoverflow.com/questions/25163598/how-do-i-reference-a-specific-issue-
↪comment-on-github) for more details.
```

PR-NEW: Better description

```
Thanks for the pull request! Please make the title of the PR descriptive so that␣
↪we can easily recall the issue it is resolving. You should state what issue (or␣
↪PR) it fixes/resolves in the description (see [here](http://scikit-learn.org/
↪dev/developers/contributing.html#contributing-pull-requests)).
```

PR-NEW: Fix #

```
Please use "Fix #issueNumber" in your PR description (and you can do it more than␣
↪once). This way the associated issue gets closed automatically when the PR is␣
↪merged. For more details, look at [this](https://github.com/blog/1506-closing-
↪issues-via-pull-requests).
```

PR-NEW or Issue: Maintenance cost

```
Every feature we include has a [maintenance cost](http://scikit-learn.org/dev/faq.
↪html#why-are-you-so-selective-on-what-algorithms-you-include-in-scikit-learn).␣
↪Our maintainers are mostly volunteers. For a new feature to be included, we␣
↪need evidence that it is often useful and, ideally, [well-established](http://
↪scikit-learn.org/dev/faq.html#what-are-the-inclusion-criteria-for-new-
↪algorithms) in the literature or in practice. That doesn't stop you␣
↪implementing it for yourself and publishing it in a separate repository, or␣
↪even [scikit-learn-contrib](https://scikit-learn-contrib.github.io).
```

PR-WIP: What's needed before merge?

```
Please clarify (perhaps as a TODO list in the PR description) what work you␣
↪believe still needs to be done before it can be reviewed for merge. When it is␣
↪ready, please prefix the PR title with `[MRG]`.
```

PR-WIP: Regression test needed

```
Please add a [non-regression test](https://en.wikipedia.org/wiki/Non-regression_
↪testing) that would fail at master but pass in this PR.
```

PR-WIP: PEP8

```
You have some [PEP8](https://www.python.org/dev/peps/pep-0008/) violations, whose␣
↪details you can see in the Circle CI `lint` job. It might be worth configuring␣
↪your code editor to check for such errors on the fly, so you can catch them␣
↪before committing.
```

PR-MRG: Patience

```
Before merging, we generally require two core developers to agree that your pull␣
↪request is desirable and ready. [Please be patient](http://scikit-learn.org/dev/
↪faq.html#why-is-my-pull-request-not-getting-any-attention), as we mostly rely␣
↪on volunteered time from busy core developers. (You are also welcome to help us␣
↪out with [reviewing other PRs](http://scikit-learn.org/dev/developers/
↪contributing.html#code-review-guidelines).)
```

PR-MRG: Add to what's new

```
Please add an entry to the change log at `doc/whats_new/v*.rst`. Like the other␣
↪entries there, please reference this pull request with `:pr:` and credit␣
↪yourself (and other contributors if applicable) with `:user:`.
```

PR: Don't change unrelated

```
Please do not change unrelated lines. It makes your contribution harder to review␣
↪and may introduce merge conflicts to other pull requests.
```

7.2.2 Debugging memory errors in Cython with valgrind

While python/numpy's built-in memory management is relatively robust, it can lead to performance penalties for some routines. For this reason, much of the high-performance code in scikit-learn in written in cython. This performance gain comes with a tradeoff, however: it is very easy for memory bugs to crop up in cython code, especially in situations where that code relies heavily on pointer arithmetic.

Memory errors can manifest themselves a number of ways. The easiest ones to debug are often segmentation faults and related glibc errors. Uninitialized variables can lead to unexpected behavior that is difficult to track down. A very useful tool when debugging these sorts of errors is valgrind.

Valgrind is a command-line tool that can trace memory errors in a variety of code. Follow these steps:

1. Install valgrind on your system.
2. Download the python valgrind suppression file: valgrind-python.supp.
3. Follow the directions in the README.valgrind file to customize your python suppressions. If you don't, you will have spurious output coming related to the python interpreter instead of your own code.
4. Run valgrind as follows:

```
$> valgrind -v --suppressions=valgrind-python.supp python my_test_script.py
```

The result will be a list of all the memory-related errors, which reference lines in the C-code generated by cython from your .pyx file. If you examine the referenced lines in the .c file, you will see comments which indicate the corresponding location in your .pyx source file. Hopefully the output will give you clues as to the source of your memory error.

For more information on valgrind and the array of options it has, see the tutorials and documentation on the valgrind web site.

7.3 Utilities for Developers

Scikit-learn contains a number of utilities to help with development. These are located in *sklearn.utils*, and include tools in a number of categories. All the following functions and classes are in the module *sklearn.utils*.

Warning: These utilities are meant to be used internally within the scikit-learn package. They are not guaranteed to be stable between versions of scikit-learn. Backports, in particular, will be removed as the scikit-learn dependencies evolve.

7.3.1 Validation Tools

These are tools used to check and validate input. When you write a function which accepts arrays, matrices, or sparse matrices as arguments, the following should be used when applicable.

- *assert_all_finite*: Throw an error if array contains NaNs or Infs.
- *as_float_array*: convert input to an array of floats. If a sparse matrix is passed, a sparse matrix will be returned.
- *check_array*: check that input is a 2D array, raise error on sparse matrices. Allowed sparse matrix formats can be given optionally, as well as allowing 1D or N-dimensional arrays. Calls *assert_all_finite* by default.

- *check_X_y*: check that X and y have consistent length, calls check_array on X, and column_or_1d on y. For multilabel classification or multitarget regression, specify multi_output=True, in which case check_array will be called on y.
- *indexable*: check that all input arrays have consistent length and can be sliced or indexed using safe_index. This is used to validate input for cross-validation.
- *validation.check_memory* checks that input is `joblib.Memory`-like, which means that it can be converted into a `sklearn.utils.Memory` instance (typically a str denoting the `cachedir`) or has the same interface.

If your code relies on a random number generator, it should never use functions like `numpy.random.random` or `numpy.random.normal`. This approach can lead to repeatability issues in unit tests. Instead, a `numpy.random.RandomState` object should be used, which is built from a `random_state` argument passed to the class or function. The function *check_random_state*, below, can then be used to create a random number generator object.

- *check_random_state*: create a `np.random.RandomState` object from a parameter `random_state`.
  - If `random_state` is `None` or `np.random`, then a randomly-initialized `RandomState` object is returned.
  - If `random_state` is an integer, then it is used to seed a new `RandomState` object.
  - If `random_state` is a `RandomState` object, then it is passed through.

For example:

```
>>> from sklearn.utils import check_random_state
>>> random_state = 0
>>> random_state = check_random_state(random_state)
>>> random_state.rand(4)
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318])
```

When developing your own scikit-learn compatible estimator, the following helpers are available.

- *validation.check_is_fitted*: check that the estimator has been fitted before calling `transform`, `predict`, or similar methods. This helper allows to raise a standardized error message across estimator.
- *validation.has_fit_parameter*: check that a given parameter is supported in the `fit` method of a given estimator.

7.3.2 Efficient Linear Algebra & Array Operations

- *extmath.randomized_range_finder*: construct an orthonormal matrix whose range approximates the range of the input. This is used in *extmath.randomized_svd*, below.
- *extmath.randomized_svd*: compute the k-truncated randomized SVD. This algorithm finds the exact truncated singular values decomposition using randomization to speed up the computations. It is particularly fast on large matrices on which you wish to extract only a small number of components.
- `arrayfuncs.cholesky_delete`: (used in *sklearn.linear_model.lars_path*) Remove an item from a cholesky factorization.
- *arrayfuncs.min_pos*: (used in `sklearn.linear_model.least_angle`) Find the minimum of the positive values within an array.
- *extmath.fast_logdet*: efficiently compute the log of the determinant of a matrix.
- *extmath.density*: efficiently compute the density of a sparse vector

- *extmath.safe_sparse_dot*: dot product which will correctly handle `scipy.sparse` inputs. If the inputs are dense, it is equivalent to `numpy.dot`.
- *extmath.weighted_mode*: an extension of `scipy.stats.mode` which allows each item to have a real-valued weight.
- *resample*: Resample arrays or sparse matrices in a consistent way. used in *shuffle*, below.
- *shuffle*: Shuffle arrays or sparse matrices in a consistent way. Used in *sklearn.cluster.k_means*.

7.3.3 Efficient Random Sampling

- *random.sample_without_replacement*: implements efficient algorithms for sampling `n_samples` integers from a population of size `n_population` without replacement.

7.3.4 Efficient Routines for Sparse Matrices

The `sklearn.utils.sparsefuncs` cython module hosts compiled extensions to efficiently process `scipy.sparse` data.

- *sparsefuncs.mean_variance_axis*: compute the means and variances along a specified axis of a CSR matrix. Used for normalizing the tolerance stopping criterion in *sklearn.cluster.KMeans*.
- *sparsefuncs_fast.inplace_csr_row_normalize_l1* and *sparsefuncs_fast.inplace_csr_row_normalize_l2*: can be used to normalize individual sparse samples to unit L1 or L2 norm as done in *sklearn.preprocessing.Normalizer*.
- *sparsefuncs.inplace_csr_column_scale*: can be used to multiply the columns of a CSR matrix by a constant scale (one scale per column). Used for scaling features to unit standard deviation in *sklearn.preprocessing.StandardScaler*.

7.3.5 Graph Routines

- *graph.single_source_shortest_path_length*: (not currently used in scikit-learn) Return the shortest path from a single source to all connected nodes on a graph. Code is adapted from networkx. If this is ever needed again, it would be far faster to use a single iteration of Dijkstra's algorithm from `graph_shortest_path`.
- *graph_shortest_path.graph_shortest_path*: (used in *sklearn.manifold.Isomap*) Return the shortest path between all pairs of connected points on a directed or undirected graph. Both the Floyd-Warshall algorithm and Dijkstra's algorithm are available. The algorithm is most efficient when the connectivity matrix is a `scipy.sparse.csr_matrix`.

7.3.6 Testing Functions

- *testing.assert_in*, *testing.assert_not_in*: Assertions for container membership. Designed for forward compatibility with Nose 1.0.
- *testing.assert_raise_message*: Assertions for checking the error raise message.
- *testing.mock_mldata_urlopen*: Mocks the urlopen function to fake requests to mldata.org. Used in tests of *sklearn.datasets*.
- *testing.all_estimators* : returns a list of all estimators in scikit-learn to test for consistent behavior and interfaces.

7.3.7 Multiclass and multilabel utility function

- *multiclass.is_multilabel*: Helper function to check if the task is a multi-label classification one.
- *multiclass.unique_labels*: Helper function to extract an ordered array of unique labels from different formats of target.

7.3.8 Helper Functions

- *gen_even_slices*: generator to create n-packs of slices going up to n. Used in *sklearn.decomposition.dict_learning* and *sklearn.cluster.k_means*.
- *safe_mask*: Helper function to convert a mask to the format expected by the numpy array or scipy sparse matrix on which to use it (sparse matrices support integer indices only while numpy arrays support both boolean masks and integer indices).
- *safe_sqr*: Helper function for unified squaring (**2) of array-likes, matrices and sparse matrices.

7.3.9 Hash Functions

- *murmurhash3_32* provides a python wrapper for the MurmurHash3_x86_32 C++ non cryptographic hash function. This hash function is suitable for implementing lookup tables, Bloom filters, Count Min Sketch, feature hashing and implicitly defined sparse random projections:

```
>>> from sklearn.utils import murmurhash3_32
>>> murmurhash3_32("some feature", seed=0) == -384616559
True

>>> murmurhash3_32("some feature", seed=0, positive=True) == 3910350737
True
```

  The sklearn.utils.murmurhash module can also be "cimported" from other cython modules so as to benefit from the high performance of MurmurHash while skipping the overhead of the Python interpreter.

7.3.10 Warnings and Exceptions

- *deprecated*: Decorator to mark a function or class as deprecated.
- *sklearn.exceptions.ConvergenceWarning*: Custom warning to catch convergence problems. Used in sklearn.covariance.graphical_lasso.

7.4 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 *A simple algorithmic trick: warm restarts* gives an example of such a trick.

7.4.1 Python, Cython or C/C++?

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 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 *svm.LinearSVC*, *svm.SVC* and *linear_model.LogisticRegression* (wrappers for liblinear and libsvm).
3. Otherwise, write an optimized version of your Python function using **Cython** directly. This strategy is used for the *linear_model.ElasticNet* and *linear_model.SGDClassifier* classes for instance.
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

    $ 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.

7.4.2 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 *Recognizing hand-written digits* example:

```
In [1]: from sklearn.decomposition import NMF

In [2]: from sklearn.datasets import load_digits

In [3]: X = load_digits().data
```

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  tottime  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 `tottime` 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  tottime  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:

```
$ pip install line_profiler
```

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

  ```
  $ 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.nmf import _nls_subproblem, NMF

In [3]: X = load_digits().data

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 solver.")
   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.

7.4.3 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:

```
$ 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:

  ```
  $ 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?`.

7.4.4 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_2/

Using OpenMP

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

```
# importing OpenMP
IF SKLEARN_OPENMP_SUPPORTED:
    cimport openmp

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

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

7.4.5 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 itself.

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:

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

# 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'.

7.4.6 Multi-core parallelism using `joblib.Parallel`

TODO: give a simple teaser example here.

Checkout the official joblib documentation:

- https://joblib.readthedocs.io

7.4.7 A simple algorithmic trick: warm restarts

See the glossary entry for warm_start

7.5 Advanced installation instructions

There are different ways to get scikit-learn installed:

- *Install an official release*. This is the best approach for most users. It will provide a stable version and pre-build packages are available for most platforms.
- Install the version of scikit-learn provided by your operating system or Python distribution. This is a quick option for those who have operating systems that distribute scikit-learn. It might not provide the latest release version.
- *Building the package from source*. This is best for users who want the latest-and-greatest features and aren't afraid of running brand-new code. This document describes how to build from source.

Note: If you wish to contribute to the project, you need to *install the latest development version*.

7.5.1 Installing nightly builds

The continuous integration servers of the scikit-learn project build, test and upload wheel packages for the most recent Python version on a nightly basis to help users test bleeding edge features or bug fixes:

```
pip install --pre -f https://sklearn-nightly.scdn8.secure.raxcdn.com scikit-learn
```

7.5.2 Building from source

In the vast majority of cases, building scikit-learn for development purposes can be done with:

```
pip install cython pytest flake8
```

Then, in the main repository:

```
pip install --editable .
```

Please read below for details and more advanced instructions.

Dependencies

Scikit-learn requires:

- Python (>= 3.5),
- NumPy (>= 1.11),
- SciPy (>= 0.17),
- Joblib (>= 0.11).

Note: For installing on PyPy, PyPy3-v5.10+, Numpy 1.14.0+, and scipy 1.1.0+ are required. For PyPy, only installation instructions with pip apply.

Building Scikit-learn also requires

- Cython >=0.28.5

- OpenMP

Note: It is possible to build scikit-learn without OpenMP support by setting the `SKLEARN_NO_OPENMP` environment variable (before cythonization). This is not recommended since it will force some estimators to run in sequential mode and their `n_jobs` parameter will be ignored.

Running tests requires

- pytest >=3.3.0

Some tests also require pandas.

Retrieving the latest code

We use Git for version control and GitHub for hosting our main repository.

You can check out the latest sources with the command:

```
git clone git://github.com/scikit-learn/scikit-learn.git
```

If you want to build a stable version, you can `git checkout <VERSION>` to get the code for that particular version, or download an zip archive of the version from github.

Once you have all the build requirements installed (see below for details), you can build and install the package in the following way.

If you run the development version, it is cumbersome to reinstall the package each time you update the sources. Therefore it's recommended that you install in editable mode, which allows you to edit the code in-place. This builds the extension in place and creates a link to the development directory (see the pip docs):

```
pip install --editable .
```

Note: This is fundamentally similar to using the command `python setup.py develop` (see the setuptool docs). It is however preferred to use pip.

Note: You will have to re-run:

```
pip install --editable .
```

every time the source code of a compiled extension is changed (for instance when switching branches or pulling changes from upstream). Compiled extensions are Cython files (ending in `.pyx` or `.pxd`).

On Unix-like systems, you can equivalently type `make in` from the top-level folder. Have a look at the `Makefile` for additional utilities.

Mac OSX

The default C compiler, Apple-clang, on Mac OSX does not directly support OpenMP. The first solution to build scikit-learn is to install another C compiler such as gcc or llvm-clang. Another solution is to enable OpenMP support on the default Apple-clang. In the following we present how to configure this second option.

You first need to install the OpenMP library:

```
brew install libomp
```

Then you need to set the following environment variables:

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
export CPPFLAGS="$CPPFLAGS -Xpreprocessor -fopenmp"
export CFLAGS="$CFLAGS -I/usr/local/opt/libomp/include"
export CXXFLAGS="$CXXFLAGS -I/usr/local/opt/libomp/include"
export LDFLAGS="$LDFLAGS -L/usr/local/opt/libomp/lib -lomp"
export DYLD_LIBRARY_PATH=/usr/local/opt/libomp/lib
```

Finally you can build the package using the standard command.

FreeBSD

The clang compiler included in FreeBSD 12.0 and 11.2 base systems does not include OpenMP support. You need to install the `openmp` library from packages (or ports):

```
sudo pkg install openmp
```

This will install header files in `/usr/local/include` and libs in `/usr/local/lib`. Since these directories are not searched by default, you can set the environment variables to these locations:

```
export CFLAGS="$CFLAGS -I/usr/local/include"
export CXXFLAGS="$CXXFLAGS -I/usr/local/include"
export LDFLAGS="$LDFLAGS -L/usr/local/lib -lomp"
export DYLD_LIBRARY_PATH=/usr/local/lib
```

Finally you can build the package using the standard command.

For the upcomming FreeBSD 12.1 and 11.3 versions, OpenMP will be included in the base system and these steps will not be necessary.

7.5.3 Installing build dependencies

Linux

Installing from source without conda requires you to have installed the scikit-learn runtime dependencies, Python development headers and a working C/C++ compiler. Under Debian-based operating systems, which include Ubuntu:

```
sudo apt-get install build-essential python3-dev python3-setuptools \
                 python3-pip
```

and then:

```
pip3 install numpy scipy cython
```

Note: In order to build the documentation and run the example code contains in this documentation you will need matplotlib:

```
pip3 install matplotlib
```

When precompiled wheels are not avalaible for your architecture, you can install the system versions:

```
sudo apt-get install cython3 python3-numpy python3-scipy python3-matplotlib
```

On Red Hat and clones (e.g. CentOS), install the dependencies using:

```
sudo yum -y install gcc gcc-c++ python-devel numpy scipy
```

Note: To use a high performance BLAS library (e.g. OpenBlas) see scipy installation instructions.

Windows

To build scikit-learn on Windows you need a working C/C++ compiler in addition to numpy, scipy and setuptools.

The building command depends on the architecture of the Python interpreter, 32-bit or 64-bit. You can check the architecture by running the following in `cmd` or `powershell` console:

```
python -c "import struct; print(struct.calcsize('P') * 8)"
```

The above commands assume that you have the Python installation folder in your PATH environment variable.

You will need Build Tools for Visual Studio 2017.

For 64-bit Python, configure the build environment with:

```
SET DISTUTILS_USE_SDK=1
"C:\Program Files (x86)\Microsoft Visual␣
↪Studio\2017\BuildTools\VC\Auxiliary\Build\vcvarsall.bat" x64
```

And build scikit-learn from this environment:

```
python setup.py install
```

Replace `x64` by `x86` to build for 32-bit Python.

Building binary packages and installers

The `.whl` package and `.exe` installers can be built with:

```
pip install wheel
python setup.py bdist_wheel bdist_wininst -b doc/logos/scikit-learn-logo.bmp
```

The resulting packages are generated in the `dist/` folder.

Using an alternative compiler

It is possible to use MinGW (a port of GCC to Windows OS) as an alternative to MSVC for 32-bit Python. Not that extensions built with mingw32 can be redistributed as reusable packages as they depend on GCC runtime libraries typically not installed on end-users environment.

To force the use of a particular compiler, pass the `--compiler` flag to the build step:

```
python setup.py build --compiler=my_compiler install
```

where `my_compiler` should be one of `mingw32` or `msvc`.

7.6 Maintainer / core-developer information

7.6.1 Before a release

1. Update authors table:

   ```
   $ cd build_tools; make authors; cd ..
   ```

   and commit.

2. Confirm any blockers tagged for the milestone are resolved, and that other issues tagged for the milestone can be postponed.

3. Ensure the change log and commits correspond (within reason!), and that the change log is reasonably well curated. Some tools for these tasks include:

   - `maint_tools/sort_whats_new.py` can put what's new entries into sections.
   - The `maint_tools/whats_missing.sh` script may be used to identify pull requests that were merged but likely missing from What's New.

Preparing a bug-fix-release

Since any commits to a released branch (e.g. 0.999.X) will automatically update the web site documentation, it is best to develop a bug-fix release with a pull request in which 0.999.X is the base. It also allows you to keep track of any tasks towards release with a TO DO list.

Most development of the bug fix release, and its documentation, should happen in master to avoid asynchrony. To select commits from master for use in the bug fix (version 0.999.3), you can use:

```
$ git checkout -b release-0.999.3 master
$ git rebase -i 0.999.X
```

Then pick the commits for release and resolve any issues, and create a pull request with 0.999.X as base. Add a commit updating `sklearn.__version__`. Additional commits can be cherry-picked into the `release-0.999.3` branch while preparing the release.

7.6.2 Making a release

1. Update docs:

   - Edit the doc/whats_new.rst file to add release title and commit statistics. You can retrieve commit statistics with:

     ```
     $ git shortlog -s 0.99.33.. | cut -f2- | sort --ignore-case | tr '\n' ';' |␣
     ↪sed 's/;/, /g;s/, $//'
     ```

   - Update the release date in whats_new.rst
   - Edit the doc/index.rst to change the 'News' entry of the front page.

- Note that these changes should be made in master and cherry-picked into the release branch.

2. On the branch for releasing, update the version number in sklearn/__init__.py, the `__version__` variable by removing `dev*` only when ready to release. On master, increment the verson in the same place (when branching for release).

3. Create the tag and push it:

```
$ git tag -a 0.999

$ git push git@github.com:scikit-learn/scikit-learn.git --tags
```

4. Create the source tarball:

   - Wipe clean your repo:

   ```
   $ git clean -xfd
   ```

   - Generate the tarball:

   ```
   $ python setup.py sdist
   ```

   The result should be in the `dist/` folder. We will upload it later with the wheels. Check that you can install it in a new virtualenv and that the tests pass.

5. Update the dependency versions and set `BUILD_COMMIT` variable to the release tag at:

   https://github.com/MacPython/scikit-learn-wheels

   Once the CI has completed successfully, collect the generated binary wheel packages and upload them to PyPI by running the following commands in the scikit-learn source folder (checked out at the release tag):

```
$ pip install -U wheelhouse_uploader twine
$ python setup.py fetch_artifacts
```

6. Check the content of the `dist/` folder: it should contain all the wheels along with the source tarball ("scikit-learn-XXX.tar.gz").

   Make sure that you do not have developer versions or older versions of the scikit-learn package in that folder.

   Upload everything at once to https://pypi.org:

```
$ twine upload dist/
```

7. For major/minor (not bug-fix release), update the symlink for `stable` in https://github.com/scikit-learn/scikit-learn.github.io:

```
$ cd /tmp
$ git clone --depth 1 --no-checkout git@github.com:scikit-learn/scikit-learn.
↪github.io.git
$ cd scikit-learn.github.io
$ echo stable > .git/info/sparse-checkout
$ git checkout master
$ ln -sf 0.999 stable
$ git push origin master
```

The following GitHub checklist might be helpful in a release PR:

```
* [ ] update news and what's new date in master and release branch
* [ ] create tag
* [ ] update dependencies and release tag at https://github.com/MacPython/scikit-
↪learn-wheels
```

```
* [ ] twine the wheels to PyPI when that's green
* [ ] https://github.com/scikit-learn/scikit-learn/releases draft
* [ ] confirm bot detected at https://github.com/conda-forge/scikit-learn-feedstock␣
↪and wait for merge
* [ ] https://github.com/scikit-learn/scikit-learn/releases publish
* [ ] announce on mailing list
* [ ] (regenerate Dash docs: https://github.com/Kapeli/Dash-User-Contributions/tree/
↪master/docsets/Scikit)
```

7.6.3 The scikit-learn.org web site

The scikit-learn web site (http://scikit-learn.org) is hosted at GitHub, but should rarely be updated manually by pushing to the https://github.com/scikit-learn/scikit-learn.github.io repository. Most updates can be made by pushing to master (for /dev) or a release branch like 0.99.X, from which Circle CI builds and uploads the documentation automatically.

7.6.4 Travis Cron jobs

From https://docs.travis-ci.com/user/cron-jobs: Travis CI cron jobs work similarly to the cron utility, they run builds at regular scheduled intervals independently of whether any commits were pushed to the repository. Cron jobs always fetch the most recent commit on a particular branch and build the project at that state. Cron jobs can run daily, weekly or monthly, which in practice means up to an hour after the selected time span, and you cannot set them to run at a specific time.

For scikit-learn, Cron jobs are used for builds that we do not want to run in each PR. As an example the build with the dev versions of numpy and scipy is run as a Cron job. Most of the time when this numpy-dev build fail, it is related to a numpy change and not a scikit-learn one, so it would not make sense to blame the PR author for the Travis failure.

The definition of what gets run in the Cron job is done in the .travis.yml config file, exactly the same way as the other Travis jobs. We use a `if: type = cron` filter in order for the build to be run only in Cron jobs.

The branch targeted by the Cron job and the frequency of the Cron job is set via the web UI at https://www.travis-ci.org/scikit-learn/scikit-learn/settings.

7.6.5 Experimental features

The *sklearn.experimental* module was introduced in 0.21 and contains experimental features / estimators that are subject to change without deprecation cycle.

To create an experimental module, you can just copy and modify the content of enable_hist_gradient_boosting.py, or enable_iterative_imputer.py.

Note that the public import path must be to a public subpackage (like `sklearn/ensemble` or `sklearn/impute`), not just a `.py` module. Also, the (private) experimental features that are imported must be in a submodule/subpackage of the public subpackage, e.g. `sklearn/ensemble/_hist_gradient_boosting/` or `sklearn/impute/_iterative.py`. This is needed so that pickles still work in the future when the features aren't experimental anymore

Please also write basic tests following those in test_enable_hist_gradient_boosting.py.

Make sure every user-facing code you write explicitly mentions that the feature is experimental, and add a `# noqa` comment to avoid pep8-related warnings:

```
# To use this experimental feature, we need to explicitly ask for it:
from sklearn.experimental import enable_hist_gradient_boosting  # noqa
from sklearn.ensemble import HistGradientBoostingRegressor
```

For the docs to render properly, please also import `enable_my_experimental_feature` in `doc/conf.py`, else sphinx won't be able to import the corresponding modules. Note that using `from sklearn.experimental import *` **does not work**.

Note that some experimental classes / functions are not included in the *`sklearn.experimental`* module: `sklearn.datasets.fetch_openml`.