scikit-learn.org

Glossary of Common Terms and API Elements — scikit-learn 0.20.0 documentation

27-34 minutes

1d¶

1d array¶

One-dimensional array. A NumPy array whose `.shape` has length 1. A vector.

2d¶

2d array¶

Two-dimensional array. A NumPy array whose `.shape` has length 2. Often represents a matrix.

API¶

Refers to both the *specific* interfaces for estimators implemented in Scikit-learn and the *generalized* conventions across types of estimators as described in this glossary and overviewed in the contributor documentation.

The specific interfaces that constitute Scikit-learn's public API are largely documented in API Reference. However we less formally consider anything as public API if none of the identifiers required to access it begins with _. We generally try to maintain backwards compatibility for all objects in the public API.

Private API, including functions, modules and methods beginning _ are not assured to be stable.

array-like¶

The most common data format for *input* to Scikit-learn estimators and functions, array-like is any type object for which `numpy.asarray` will produce an array of appropriate shape (usually 1 or 2-dimensional) of appropriate dtype (usually numeric).

This includes:

- a numpy array
- a list of numbers

- a list of length-k lists of numbers for some fixed length k
- a `pandas.DataFrame` with all columns numeric
- a numeric `pandas.Series`

  It excludes:

- a sparse matrix
- an iterator
- a generator

  Note that *output* from scikit-learn estimators and functions (e.g. predictions) should generally be arrays or sparse matrices, or lists thereof (as in multi-output `tree.DecisionTreeClassifier`’s `predict_proba`). An estimator where `predict()` returns a list or a *pandas.Series* is not valid.

attribute¶
attributes¶

We mostly use attribute to refer to how model information is stored on an estimator during fitting. Any public attribute stored on an estimator instance is required to begin with an alphabetic character and end in a single underscore if it is set in fit or partial_fit. These are what is documented under an estimator’s *Attributes* documentation. The information stored in attributes is usually either: sufficient statistics used for prediction or transformation; transductive outputs such as labels_ or embedding_; or diagnostic data, such as feature_importances_. Common attributes are listed below.

A public attribute may have the same name as a constructor parameter, with a _ appended. This is used to store a validated or estimated version of the user’s input. For example, `decomposition.PCA` is constructed with an `n_components` parameter. From this, together with other parameters and the data, PCA estimates the attribute `n_components_`.

Further private attributes used in prediction/transformation/etc. may also be set when fitting. These begin with a single underscore and are not assured to be stable for public access.

A public attribute on an estimator instance that does not end in an underscore should be the stored, unmodified value of an `__init__` parameter of the same name. Because of this equivalence, these are documented under an estimator’s *Parameters* documentation.

backwards compatibility¶

We generally try to maintain backwards compatibility (i.e. interfaces and behaviors may be extended but not changed or removed) from release to release but this comes with some exceptions:

Public API only
: The behaviour of objects accessed through private identifiers (those beginning _) may be changed arbitrarily between versions.

As documented
: We will generally assume that the users have adhered to the documented parameter types and ranges. If the documentation asks for a list and the user gives a tuple, we do not assure consistent behavior from version to version.

Deprecation
: Behaviors may change following a deprecation period (usually two releases long). Warnings are issued using Python's `warnings` module.

Keyword arguments
: We may sometimes assume that all optional parameters (other than X and y to fit and similar methods) are passed as keyword arguments only and may be positionally reordered.

Bug fixes and enhancements
: Bug fixes and – less often – enhancements may change the behavior of estimators, including the predictions of an estimator trained on the same data and random_state. When this happens, we attempt to note it clearly in the changelog.

Serialization
: We make no assurances that pickling an estimator in one version will allow it to be unpickled to an equivalent model in the subsequent version. (For estimators in the sklearn package, we issue a warning when this unpickling is attempted, even if it may happen to work.) See Security & maintainability limitations.

`utils.estimator_checks.check_estimator`
: We provide limited backwards compatibility assurances for the estimator checks: we may add extra requirements on estimators tested with this function, usually when these were informally assumed but not formally tested.

Despite this informal contract with our users, the software is provided as is, as stated in the licence. When a release inadvertently introduces changes that are not backwards compatible, these are known as software regressions.

callable¶

A function, class or an object which implements the `__call__` method; anything that returns True when the argument of callable().

categorical feature¶

A categorical or nominal feature is one that has a finite set of discrete values across the population of data. These are commonly represented as columns of integers or strings. Strings will be rejected by most scikit-learn estimators, and integers will be treated as ordinal or count-valued. For the use with most estimators, categorical variables should be one-hot encoded. Notable exceptions include tree-based models such as random forests and gradient boosting models that often work better and faster with integer-coded categorical variables. `OrdinalEncoder` helps encoding string-valued categorical features as ordinal integers, and `OneHotEncoder` can be used to one-hot encode categorical features. See also Encoding categorical features and the http://contrib.scikit-learn.org/categorical-encoding package for tools related to encoding categorical features.

clone¶

cloned¶

To copy an estimator instance and create a new one with identical parameters, but without any fitted attributes, using `clone`.

When `fit` is called, a meta-estimator usually clones a wrapped estimator instance before fitting the cloned instance. (Exceptions, for legacy reasons, include `Pipeline` and `FeatureUnion`.)

common tests¶

This refers to the tests run on almost every estimator class in Scikit-learn to check they comply with basic API conventions. They are available for external use through `utils.estimator_checks.check_estimator`, with most of the implementation in `sklearn/utils/estimator_checks.py`.

Note: Some exceptions to the common testing regime are currently hard-coded into the library, but we hope to replace this by marking exceptional behaviours on the estimator using semantic estimator tags.

deprecation¶

We use deprecation to slowly violate our backwards compatibility assurances, usually to to:

- change the default value of a parameter; or
- remove a parameter, attribute, method, class, etc.

We will ordinarily issue a warning when a deprecated element is used, although there may be limitations to this. For instance, we will raise a warning when someone sets a parameter that has been deprecated, but may not when they access that parameter's attribute on the estimator instance.

See the Contributors' Guide.

dimensionality¶

May be used to refer to the number of features (i.e. n_features), or columns in a 2d feature matrix. Dimensions are, however, also used to refer to the length of a NumPy array's shape, distinguishing a 1d array from a 2d matrix.

docstring¶

The embedded documentation for a module, class, function, etc., usually in code as a string at the beginning of the object's definition, and accessible as the object's `__doc__` attribute.

We try to adhere to PEP257, and follow NumpyDoc conventions.

double underscore¶

double underscore notation¶

When specifying parameter names for nested estimators, `__` may be used to separate between parent and child in some contexts. The most common use is when setting parameters through a meta-estimator with set_params and hence in specifying a search grid in parameter search. See parameter. It is also used in `pipeline.Pipeline.fit` for passing sample properties to the `fit` methods of estimators in the pipeline.

dtype¶

data type¶

NumPy arrays assume a homogeneous data type throughout, available in the `.dtype` attribute of an array (or sparse matrix). We generally assume simple data types for scikit-learn data: float or integer. We may support object or string data types for arrays before encoding or vectorizing. Our estimators do not work with struct arrays, for instance.

TODO: Mention efficiency and precision issues; casting policy.

duck typing¶

We try to apply duck typing to determine how to handle some input values (e.g. checking whether a given estimator is a classifier). That is, we avoid using `isinstance` where possible, and rely on the presence or absence of attributes to determine an object's behaviour. Some nuance is required when following this approach:

- For some estimators, an attribute may only be available once it is fitted. For instance, we cannot a priori determine if predict_proba is available in a grid search where the grid includes alternating between a probabilistic and a non-probabilistic predictor in the final step of the pipeline. In the following, we can only determine if `clf` is probabilistic after fitting it on some data:

  >>>

  ```
  >>> from sklearn.model_selection import
  GridSearchCV
  >>> from sklearn.linear_model import
  SGDClassifier
  >>> clf = GridSearchCV(SGDClassifier(),
  ...                     param_grid={'loss':
  ['log', 'hinge']})
  ```

  This means that we can only check for duck-typed attributes after fitting, and that we must be careful to make meta-estimators only present attributes according to the state of the underlying estimator after fitting.

- Checking if an attribute is present (using `hasattr`) is in general just as expensive as getting the attribute (`getattr` or dot notation). In some cases, getting the attribute may indeed be expensive (e.g. for some implementations of feature_importances_, which may suggest this is an API design flaw). So code which does `hasattr` followed by `getattr` should be avoided; `getattr` within a try-except block is preferred.

- For determining some aspects of an estimator's expectations or support for some feature, we use estimator tags instead of duck typing.

early stopping¶

This consists in stopping an iterative optimization method before the convergence of the training loss, to avoid over-fitting. This is generally done by monitoring the generalization score on a validation set. When available, it is activated through the parameter `early_stopping` or by setting a positive n_iter_no_change.

estimator instance¶

We sometimes use this terminology to distinguish an estimator class from a constructed instance. For example, in the following, `cls` is an estimator class, while `est1` and `est2` are instances:

```
cls = RandomForestClassifier
est1 = cls()
est2 = RandomForestClassifier()
```

examples¶

We try to give examples of basic usage for most functions and classes in the API:

- as doctests in their docstrings (i.e. within the `sklearn/` library code itself).
- as examples in the example gallery rendered (using sphinx-gallery) from scripts in the `examples/` directory, exemplifying key features or parameters of the estimator/function. These should also be referenced from the User Guide.
- sometimes in the User Guide (built from `doc/`) alongside a technical description of the estimator.

evaluation metric¶

evaluation metrics¶

Evaluation metrics give a measure of how well a model performs. We may use this term specifically to refer to the functions in `metrics` (disregarding `metrics.pairwise`), as distinct from the score method and the scoring API used in cross validation. See Model evaluation: quantifying the quality of predictions.

These functions usually accept a ground truth (or the raw data where the metric evaluates clustering without a ground truth) and a prediction, be it the output of predict (`y_pred`), of predict_proba (`y_proba`), or of an arbitrary score function including decision_function (`y_score`). Functions are usually named to end with `_score` if a greater score indicates a better model, and `_loss` if a lesser score indicates a better model. This diversity of interface motivates the scoring API.

Note that some estimators can calculate metrics that are not included in `metrics` and are estimator-specific, notably model likelihoods.

estimator tags¶

A proposed feature (e.g. #8022) by which the capabilities of an estimator are described through a set of semantic tags. This would enable some runtime behaviors based on estimator inspection, but it also allows each estimator to be tested for appropriate invariances while being excepted from other common tests.

Some aspects of estimator tags are currently determined through the duck typing of methods like `predict_proba` and through some special attributes on estimator objects:

`_estimator_type`¶

This string-valued attribute identifies an estimator as being a classifier, regressor, etc. It is set by mixins such as `base.ClassifierMixin`, but needs to be more explicitly adopted on a meta-estimator. Its value should usually be checked by way of a helper such as `base.is_classifier`.

`_pairwise`¶

This boolean attribute indicates whether the data (`X`) passed to `fit` and similar methods consists of pairwise measures over samples rather than a feature representation for each sample. It is usually `True` where an estimator has a `metric` or `affinity` or `kernel` parameter with value 'precomputed'. Its primary purpose is that when a meta-estimator extracts a sub-sample of data intended for a pairwise estimator, the data needs to be indexed on both axes, while other data is indexed only on the first axis.

feature¶
features¶
feature vector¶

In the abstract, a feature is a function (in its mathematical sense) mapping a sampled object to a numeric or categorical quantity. "Feature" is also commonly used to refer to these quantities, being the individual elements of a vector representing a sample. In a data matrix, features are represented as columns: each column contains the result of applying a feature function to a set of samples.

Elsewhere features are known as attributes, predictors, regressors, or independent variables.

Nearly all estimators in scikit-learn assume that features are numeric, finite and not missing, even when they have semantically distinct domains and distributions (categorical, ordinal, count-valued, real-valued, interval). See also categorical feature and missing values.

`n_features` indicates the number of features in a dataset.

fitting¶

Calling fit (or fit_transform, fit_predict, etc.) on an estimator.

fitted¶

The state of an estimator after fitting.

There is no conventional procedure for checking if an estimator is fitted. However, an estimator that is not fitted:

- should raise `exceptions.NotFittedError` when a prediction method (predict, transform, etc.) is called. (`utils.validation.check_is_fitted` is used internally for this purpose.)
- should not have any attributes beginning with an alphabetic character and ending with an underscore. (Note that a descriptor for the attribute may still be present on the class, but hasattr should return False)

function¶

We provide ad hoc function interfaces for many algorithms, while estimator classes provide a more consistent interface.

In particular, Scikit-learn may provide a function interface that fits a model to some data and returns the learnt model parameters, as in `linear_model.enet_path`. For transductive models, this also returns the embedding or cluster labels, as in `manifold.spectral_embedding` or `cluster.dbscan`. Many preprocessing transformers also provide a function interface, akin to calling fit_transform, as in `preprocessing.maxabs_scale`. Users should be careful to avoid data leakage when making use of these `fit_transform`-equivalent functions.

We do not have a strict policy about when to or when not to provide function forms of estimators, but maintainers should consider consistency with existing interfaces, and whether providing a function would lead users astray from best practices (as regards data leakage, etc.)

gallery¶
: See examples.

hyperparameter¶
hyper-parameter¶
: See parameter.

impute¶
imputation¶
: Most machine learning algorithms require that their inputs have no missing values, and will not work if this requirement is violated. Algorithms that attempt to fill in (or impute) missing values are referred to as imputation algorithms.

indexable¶
: An array-like, sparse matrix, pandas DataFrame or sequence (usually a list).

induction¶
inductive¶
: Inductive (contrasted with transductive) machine learning builds a model of some data that can then be applied to new instances. Most estimators in Scikit-learn are inductive, having predict and/or transform methods.

joblib¶
: A Python library (http://joblib.readthedocs.io) used in Scikit-learn to facilite simple parallelism and caching. Joblib is oriented towards efficiently working with numpy arrays, such as through use of memory mapping. See Parallel and distributed computing for more information.

label indicator matrix¶
multilabel indicator matrix¶
multilabel indicator matrices¶
: The format used to represent multilabel data, where each row of a 2d array or sparse matrix corresponds to a sample, each column corresponds to a class, and each element is 1 if the sample is labeled with the class and 0 if not.

leakage¶
data leakage¶
: A problem in cross validation where generalization performance can be over-estimated since knowledge of the test data was inadvertently included in training a model. This is

a risk, for instance, when applying a transformer to the entirety of a dataset rather than each training portion in a cross validation split.

We aim to provide interfaces (such as `pipeline` and `model_selection`) that shield the user from data leakage.

memmapping¶

memory map¶

memory mapping¶

A memory efficiency strategy that keeps data on disk rather than copying it into main memory. Memory maps can be created for arrays that can be read, written, or both, using `numpy.memmap`. When using joblib to parallelize operations in Scikit-learn, it may automatically memmap large arrays to reduce memory duplication overhead in multiprocessing.

missing values¶

Most Scikit-learn estimators do not work with missing values. When they do (e.g. in `impute.SimpleImputer`), NaN is the preferred representation of missing values in float arrays. If the array has integer dtype, NaN cannot be represented. For this reason, we support specifying another `missing_values` value when imputation or learning can be performed in integer space. Unlabeled data is a special case of missing values in the target.

`n_features`¶

The number of features.

`n_outputs`¶

The number of outputs in the target.

`n_samples`¶

The number of samples.

`n_targets`¶

Synonym for n_outputs.

narrative docs¶

narrative documentation¶

An alias for User Guide, i.e. documentation written in `doc/modules/`. Unlike the API reference provided through docstrings, the User Guide aims to:

- group tools provided by Scikit-learn together thematically or in terms of usage;
- motivate why someone would use each particular tool, often through comparison;

- provide both intuitive and technical descriptions of tools;
- provide or link to examples of using key features of a tool.

np¶

A shorthand for Numpy due to the conventional import statement:

online learning¶

Where a model is iteratively updated by receiving each batch of ground truth targets soon after making predictions on corresponding batch of data. Intrinsically, the model must be usable for prediction after each batch. See partial_fit.

out-of-core¶

An efficiency strategy where not all the data is stored in main memory at once, usually by performing learning on batches of data. See partial_fit.

outputs¶

Individual scalar/categorical variables per sample in the target. For example, in multilabel classification each possible label corresponds to a binary output. Also called *responses*, *tasks* or *targets*. See multiclass multioutput and continuous multioutput.

pair¶

A tuple of length two.

parameter¶

parameters¶

param¶

params¶

We mostly use *parameter* to refer to the aspects of an estimator that can be specified in its construction. For example, `max_depth` and `random_state` are parameters of `RandomForestClassifier`. Parameters to an estimator's constructor are stored unmodified as attributes on the estimator instance, and conventionally start with an alphabetic character and end with an alphanumeric character. Each estimator's constructor parameters are described in the estimator's docstring.

We do not use parameters in the statistical sense, where parameters are values that specify a model and can be estimated from data. What we call parameters might be what statisticians call hyperparameters to the model: aspects for configuring model structure that are often not directly learnt from data. However, our parameters are also used to prescribe modeling operations that do not affect the learnt model, such as n_jobs for controlling parallelism.

When talking about the parameters of a meta-estimator, we may also be including the parameters of the estimators wrapped by the meta-estimator. Ordinarily, these nested parameters are denoted by using a double underscore (__) to separate between the estimator-as-parameter and its parameter. Thus `clf = BaggingClassifier(base_estimator=DecisionTreeClassifier(max_deptł` has a deep parameter `base_estimator__max_depth` with value `3`, which is accessible with `clf.base_estimator.max_depth` or `clf.get_params()['base_estimator__max_depth']`.

The list of parameters and their current values can be retrieved from an estimator instance using its get_params method.

Between construction and fitting, parameters may be modified using set_params. To enable this, parameters are not ordinarily validated or altered when the estimator is constructed, or when each parameter is set. Parameter validation is performed when fit is called.

Common parameters are listed below.

pairwise metric¶
pairwise metrics¶

In its broad sense, a pairwise metric defines a function for measuring similarity or dissimilarity between two samples (with each ordinarily represented as a feature vector). We particularly provide implementations of distance metrics (as well as improper metrics like Cosine Distance) through `metrics.pairwise_distances`, and of kernel functions (a constrained class of similarity functions) in `metrics.pairwise_kernels`. These can compute pairwise distance matrices that are symmetric and hence store data redundantly.

See also precomputed and metric.

Note that for most distance metrics, we rely on implementations from `scipy.spatial.distance`, but may reimplement for efficiency in our context. The `neighbors` module also duplicates some metric implementations for integration with efficient binary tree search data structures.

pd¶

A shorthand for Pandas due to the conventional import statement:

precomputed¶

Where algorithms rely on pairwise metrics, and can be computed from pairwise metrics alone, we often allow the user to specify that the X provided is already in the pairwise (dis)similarity space, rather than in a feature space. That is, when passed to fit, it is a square, symmetric matrix, with each vector indicating (dis)similarity to every sample, and when passed to prediction/transformation methods, each row corresponds to a testing sample and each column to a training sample.

Use of precomputed X is usually indicated by setting a `metric`, `affinity` or `kernel` parameter to the string 'precomputed'. An estimator should mark itself as being _pairwise if this is the case.

rectangular¶

Data that can be represented as a matrix with samples on the first axis and a fixed, finite set of features on the second is called rectangular.

This term excludes samples with non-vectorial structure, such as text, an image of arbitrary size, a time series of arbitrary length, a set of vectors, etc. The purpose of a vectorizer is to produce rectangular forms of such data.

sample¶
samples¶

We usually use this term as a noun to indicate a single feature vector. Elsewhere a sample is called an instance, data point, or observation. `n_samples` indicates the number of samples in a dataset, being the number of rows in a data array X.

sample property¶
sample properties¶

A sample property is data for each sample (e.g. an array of length n_samples) passed to an estimator method or a similar function, alongside but distinct from the features (`X`) and target (`y`). The most prominent example is sample_weight; see others at Data and sample properties.

As of version 0.19 we do not have a consistent approach to handling sample properties and their routing in meta-estimators, though a `fit_params` parameter is often used.

scikit-learn-contrib¶

A venue for publishing Scikit-learn-compatible libraries that are broadly authorized by the core developers and the contrib community, but not maintained by the core developer team. See http://scikit-learn-contrib.github.io.

semi-supervised¶

semi-supervised learning¶

semisupervised¶

Learning where the expected prediction (label or ground truth) is only available for some samples provided as training data when fitting the model. We conventionally apply the label `-1` to unlabeled samples in semi-supervised classification.

sparse matrix¶

A representation of two-dimensional numeric data that is more memory efficient the corresponding dense numpy array where almost all elements are zero. We use the `scipy.sparse` framework, which provides several underlying sparse data representations, or *formats*. Some formats are more efficient than others for particular tasks, and when a particular format provides especial benefit, we try to document this fact in Scikit-learn parameter descriptions.

Some sparse matrix formats (notably CSR, CSC, COO and LIL) distinguish between *implicit* and *explicit* zeros. Explicit zeros are stored (i.e. they consume memory in a `data` array) in the data structure, while implicit zeros correspond to every element not otherwise defined in explicit storage.

Two semantics for sparse matrices are used in Scikit-learn:

matrix semantics

The sparse matrix is interpreted as an array with implicit and explicit zeros being interpreted as the number 0. This is the interpretation most often adopted, e.g. when sparse matrices are used for feature matrices or multilabel indicator matrices.

graph semantics

As with `scipy.sparse.csgraph`, explicit zeros are interpreted as the number 0, but implicit zeros indicate a masked or absent value, such as the absence of an edge between two vertices of a graph, where an explicit value indicates an edge's weight. This interpretation is adopted to represent connectivity in clustering, in representations of nearest neighborhoods (e.g.

`neighbors.kneighbors_graph`), and for precomputed distance representation where only distances in the neighborhood of each point are required.

When working with sparse matrices, we assume that it is sparse for a good reason, and avoid writing code that densifies a user-provided sparse matrix, instead maintaining sparsity or raising an error if not possible (i.e. if an estimator does not / cannot support sparse matrices).

supervised¶
supervised learning¶

Learning where the expected prediction (label or ground truth) is available for each sample when fitting the model, provided as y. This is the approach taken in a classifier or regressor among other estimators.

target¶
targets¶

The *dependent variable* in supervised (and semisupervised) learning, passed as y to an estimator's fit method. Also known as *dependent variable*, *outcome variable*, *response variable*, *ground truth* or *label*. Scikit-learn works with targets that have minimal structure: a class from a finite set, a finite real-valued number, multiple classes, or multiple numbers. See Target Types.

transduction¶
transductive¶

A transductive (contrasted with inductive) machine learning method is designed to model a specific dataset, but not to apply that model to unseen data. Examples include `manifold.TSNE`, `cluster.AgglomerativeClustering` and `neighbors.LocalOutlierFactor`.

unlabeled¶
unlabeled data¶

Samples with an unknown ground truth when fitting; equivalently, missing values in the target. See also semisupervised and unsupervised learning.

unsupervised¶
unsupervised learning¶

Learning where the expected prediction (label or ground truth) is not available for each sample when fitting the model, as in clusterers and outlier detectors. Unsupervised estimators ignore any y passed to fit.