CHAPTER

FOUR

GLOSSARY OF COMMON TERMS AND API ELEMENTS

This glossary hopes to definitively represent the tacit and explicit conventions applied in Scikit-learn and its API, while providing a reference for users and contributors. It aims to describe the concepts and either detail their corresponding API or link to other relevant parts of the documentation which do so. By linking to glossary entries from the API Reference and User Guide, we may minimize redundancy and inconsistency.

We begin by listing general concepts (and any that didn't fit elsewhere), but more specific sets of related terms are listed below: *Class APIs and Estimator Types*, *Target Types*, *Methods*, *Parameters*, *Attributes*, *Data and sample properties*.

4.1 General Concepts

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 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 (https://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:

```
import numpy as np
```

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_depth=3))` 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:

```
import pandas as pd
```

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 https://scikit-learn-contrib.github.io.

scikit-learn enhancement proposals

SLEP

SLEPs Changes to the API principles and changes to dependencies or supported versions happen via a *SLEP* and follows the decision-making process outlined in *Scikit-learn governance and decision-making*. For all votes, a proposal must have been made public and discussed before the vote. Such proposal must be a consolidated document, in the form of a 'Scikit-Learn Enhancement Proposal' (SLEP), rather than a long discussion on an issue. A SLEP must be submitted as a pull-request to enhancement proposals using the SLEP template.

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*.

4.2 Class APIs and Estimator Types

classifier

classifiers A *supervised* (or *semi-supervised*) *predictor* with a finite set of discrete possible output values.

A classifier supports modeling some of *binary*, *multiclass*, *multilabel*, or *multiclass multioutput* targets. Within scikit-learn, all classifiers support multi-class classification, defaulting to using a one-vs-rest strategy over the binary classification problem.

Classifiers must store a *classes_* attribute after fitting, and usually inherit from `base.ClassifierMixin`, which sets their *_estimator_type* attribute.

A classifier can be distinguished from other estimators with `is_classifier`.

A classifier must implement:

- *fit*
- *predict*
- *score*

It may also be appropriate to implement *decision_function*, *predict_proba* and *predict_log_proba*.

clusterer

clusterers A *unsupervised predictor* with a finite set of discrete output values.

A clusterer usually stores *labels_* after fitting, and must do so if it is *transductive*.

A clusterer must implement:

- *fit*
- *fit_predict* if *transductive*
- *predict* if *inductive*

density estimator TODO

estimator

estimators An object which manages the estimation and decoding of a model. The model is estimated as a deterministic function of:

- *parameters* provided in object construction or with *set_params*;
- the global `numpy.random` random state if the estimator's *random_state* parameter is set to None; and
- any data or *sample properties* passed to the most recent call to *fit*, *fit_transform* or *fit_predict*, or data similarly passed in a sequence of calls to *partial_fit*.

The estimated model is stored in public and private *attributes* on the estimator instance, facilitating decoding through prediction and transformation methods.

Estimators must provide a *fit* method, and should provide *set_params* and *get_params*, although these are usually provided by inheritance from `base.BaseEstimator`.

The core functionality of some estimators may also be available as a *function*.

feature extractor

feature extractors A *transformer* which takes input where each sample is not represented as an *array-like* object of fixed length, and produces an *array-like* object of *features* for each sample (and thus a 2-dimensional array-like for a set of samples). In other words, it (lossily) maps a non-rectangular data representation into *rectangular* data.

Feature extractors must implement at least:

- *fit*
- *transform*
- *get_feature_names*

meta-estimator

meta-estimators

metaestimator

metaestimators An *estimator* which takes another estimator as a parameter. Examples include `pipeline.Pipeline`, `model_selection.GridSearchCV`, `feature_selection.SelectFromModel` and `ensemble.BaggingClassifier`.

In a meta-estimator's *fit* method, any contained estimators should be *cloned* before they are fit (although FIXME: Pipeline and FeatureUnion do not do this currently). An exception to this is that an estimator may explicitly document that it accepts a prefitted estimator (e.g. using `prefit=True` in `feature_selection.SelectFromModel`). One known issue with this is that the prefitted estimator will lose its model if the meta-estimator is cloned. A meta-estimator should have `fit` called before prediction, even if all contained estimators are prefitted.

In cases where a meta-estimator's primary behaviors (e.g. *predict* or *transform* implementation) are functions of prediction/transformation methods of the provided *base estimator* (or multiple base estimators), a meta-estimator should provide at least the standard methods provided by the base estimator. It may not be possible to identify which methods are provided by the underlying estimator until the meta-estimator has been *fitted* (see also *duck typing*), for which `utils.metaestimators.if_delegate_has_method` may help. It should also provide (or modify) the *estimator tags* and *classes_* attribute provided by the base estimator.

Meta-estimators should be careful to validate data as minimally as possible before passing it to an underlying estimator. This saves computation time, and may, for instance, allow the underlying estimator to easily work with data that is not *rectangular*.

outlier detector

outlier detectors An *unsupervised* binary *predictor* which models the distinction between core and outlying samples.

Outlier detectors must implement:

- *fit*
- *fit_predict* if *transductive*
- *predict* if *inductive*

Inductive outlier detectors may also implement *decision_function* to give a normalized inlier score where outliers have score below 0. *score_samples* may provide an unnormalized score per sample.

predictor

predictors An *estimator* supporting *predict* and/or *fit_predict*. This encompasses *classifier*, *regressor*, *outlier detector* and *clusterer*.

In statistics, "predictors" refers to *features*.

regressor

regressors A *supervised* (or *semi-supervised*) *predictor* with *continuous* output values.

Regressors usually inherit from `base.RegressorMixin`, which sets their *_estimator_type* attribute.

A regressor can be distinguished from other estimators with `is_regressor`.

A regressor must implement:

- *fit*
- *predict*
- *score*

transformer

transformers An estimator supporting *transform* and/or *fit_transform*. A purely *transductive* transformer, such as `manifold.TSNE`, may not implement `transform`.

vectorizer

vectorizers See *feature extractor*.

There are further APIs specifically related to a small family of estimators, such as:

cross-validation splitter

CV splitter

cross-validation generator A non-estimator family of classes used to split a dataset into a sequence of train and test portions (see *Cross-validation: evaluating estimator performance*), by providing *split* and *get_n_splits* methods. Note that unlike estimators, these do not have *fit* methods and do not provide *set_params* or *get_params*. Parameter validation may be performed in `__init__`.

cross-validation estimator An estimator that has built-in cross-validation capabilities to automatically select the best hyper-parameters (see the *User Guide*). Some example of cross-validation estimators are `ElasticNetCV` and `LogisticRegressionCV`. Cross-validation estimators are named `EstimatorCV` and tend to be roughly equivalent to `GridSearchCV(Estimator(), ...)`. The advantage of using a cross-validation estimator over the canonical `Estimator` class along with *grid search* is that they can take advantage of warm-starting by reusing precomputed results in the previous steps of the cross-validation process. This generally leads to speed improvements. An exception is the `RidgeCV` class, which can instead perform efficient Leave-One-Out CV.

scorer A non-estimator callable object which evaluates an estimator on given test data, returning a number. Unlike *evaluation metrics*, a greater returned number must correspond with a *better* score. See *The scoring parameter: defining model evaluation rules*.

Further examples:

- `neighbors.DistanceMetric`
- `gaussian_process.kernels.Kernel`
- `tree.Criterion`

4.3 Target Types

binary A classification problem consisting of two classes. A binary target may represented as for a *multiclass* problem but with only two labels. A binary decision function is represented as a 1d array.

Semantically, one class is often considered the "positive" class. Unless otherwise specified (e.g. using pos_label in *evaluation metrics*), we consider the class label with the greater value (numerically or lexicographically) as the positive class: of labels [0, 1], 1 is the positive class; of [1, 2], 2 is the positive class; of ['no', 'yes'], 'yes' is the positive class; of ['no', 'YES'], 'no' is the positive class. This affects the output of *decision_function*, for instance.

Note that a dataset sampled from a multiclass `y` or a continuous `y` may appear to be binary.

type_of_target will return 'binary' for binary input, or a similar array with only a single class present.

continuous A regression problem where each sample's target is a finite floating point number, represented as a 1-dimensional array of floats (or sometimes ints).

type_of_target will return 'continuous' for continuous input, but if the data is all integers, it will be identified as 'multiclass'.

continuous multioutput

multioutput continuous A regression problem where each sample's target consists of `n_outputs` *outputs*, each one a finite floating point number, for a fixed int `n_outputs > 1` in a particular dataset.

Continuous multioutput targets are represented as multiple *continuous* targets, horizontally stacked into an array of shape `(n_samples, n_outputs)`.

type_of_target will return 'continuous-multioutput' for continuous multioutput input, but if the data is all integers, it will be identified as 'multiclass-multioutput'.

multiclass A classification problem consisting of more than two classes. A multiclass target may be represented as a 1-dimensional array of strings or integers. A 2d column vector of integers (i.e. a single output in *multioutput* terms) is also accepted.

We do not officially support other orderable, hashable objects as class labels, even if estimators may happen to work when given classification targets of such type.

For semi-supervised classification, *unlabeled* samples should have the special label -1 in `y`.

Within sckit-learn, all estimators supporting binary classification also support multiclass classification, using One-vs-Rest by default.

A *preprocessing.LabelEncoder* helps to canonicalize multiclass targets as integers.

type_of_target will return 'multiclass' for multiclass input. The user may also want to handle 'binary' input identically to 'multiclass'.

multiclass multioutput

multioutput multiclass A classification problem where each sample's target consists of `n_outputs` *outputs*, each a class label, for a fixed int `n_outputs > 1` in a particular dataset. Each output has a fixed set of available classes, and each sample is labelled with a class for each output. An output may be binary or multiclass, and in the case where all outputs are binary, the target is *multilabel*.

Multiclass multioutput targets are represented as multiple *multiclass* targets, horizontally stacked into an array of shape `(n_samples, n_outputs)`.

XXX: For simplicity, we may not always support string class labels for multiclass multioutput, and integer class labels should be used.

`multioutput` provides estimators which estimate multi-output problems using multiple single-output estimators. This may not fully account for dependencies among the different outputs, which methods natively handling the multioutput case (e.g. decision trees, nearest neighbors, neural networks) may do better.

type_of_target will return 'multiclass-multioutput' for multiclass multioutput input.

multilabel A *multiclass multioutput* target where each output is *binary*. This may be represented as a 2d (dense) array or sparse matrix of integers, such that each column is a separate binary target, where positive labels are indicated with 1 and negative labels are usually -1 or 0. Sparse multilabel targets are not supported everywhere that dense multilabel targets are supported.

Semantically, a multilabel target can be thought of as a set of labels for each sample. While not used internally, *preprocessing.MultiLabelBinarizer* is provided as a utility to convert from a list of sets representation to a 2d array or sparse matrix. One-hot encoding a multiclass target with *preprocessing.LabelBinarizer* turns it into a multilabel problem.

`type_of_target` will return 'multilabel-indicator' for multilabel input, whether sparse or dense.

multioutput

multi-output A target where each sample has multiple classification/regression labels. See *multiclass multioutput* and *continuous multioutput*. We do not currently support modelling mixed classification and regression targets.

4.4 Methods

`decision_function` In a fitted *classifier* or *outlier detector*, predicts a "soft" score for each sample in relation to each class, rather than the "hard" categorical prediction produced by *predict*. Its input is usually only some observed data, *X*.

If the estimator was not already *fitted*, calling this method should raise a `exceptions.NotFittedError`.

Output conventions:

binary classification A 1-dimensional array, where values strictly greater than zero indicate the positive class (i.e. the last class in *classes_*).

multiclass classification A 2-dimensional array, where the row-wise arg-maximum is the predicted class. Columns are ordered according to *classes_*.

multilabel classification Scikit-learn is inconsistent in its representation of multilabel decision functions. Some estimators represent it like multiclass multioutput, i.e. a list of 2d arrays, each with two columns. Others represent it with a single 2d array, whose columns correspond to the individual binary classification decisions. The latter representation is ambiguously identical to the multiclass classification format, though its semantics differ: it should be interpreted, like in the binary case, by thresholding at 0.

TODO: This gist higlights the use of the different formats for multilabel.

multioutput classification A list of 2d arrays, corresponding to each multiclass decision function.

outlier detection A 1-dimensional array, where a value greater than or equal to zero indicates an inlier.

`fit` The `fit` method is provided on every estimator. It usually takes some *samples* `X`, *targets* `y` if the model is supervised, and potentially other *sample properties* such as *sample_weight*. It should:

- clear any prior *attributes* stored on the estimator, unless *warm_start* is used;
- validate and interpret any *parameters*, ideally raising an error if invalid;
- validate the input data;
- estimate and store model attributes from the estimated parameters and provided data; and
- return the now *fitted* estimator to facilitate method chaining.

Target Types describes possible formats for `y`.

`fit_predict` Used especially for *unsupervised*, *transductive* estimators, this fits the model and returns the predictions (similar to *predict*) on the training data. In clusterers, these predictions are also stored in the *labels_* attribute, and the output of `.fit_predict(X)` is usually equivalent to `.fit(X).predict(X)`. The parameters to `fit_predict` are the same as those to `fit`.

`fit_transform` A method on *transformers* which fits the estimator and returns the transformed training data. It takes parameters as in *fit* and its output should have the same shape as calling `.fit(X, ...).transform(X)`. There are nonetheless rare cases where `.fit_transform(X, ...)` and `.fit(X, ...).transform(X)` do not return the same value, wherein training data needs to be handled differently (due to model blending in stacked ensembles, for instance; such cases should be clearly documented). *Transductive* transformers may also provide `fit_transform` but not *transform*.

One reason to implement `fit_transform` is that performing `fit` and `transform` separately would be less efficient than together. *base.TransformerMixin* provides a default implementation, providing a consistent interface across transformers where `fit_transform` is or is not specialised.

In *inductive* learning – where the goal is to learn a generalised model that can be applied to new data – users should be careful not to apply `fit_transform` to the entirety of a dataset (i.e. training and test data together) before further modelling, as this results in *data leakage*.

get_feature_names Primarily for *feature extractors*, but also used for other transformers to provide string names for each column in the output of the estimator's *transform* method. It outputs a list of strings, and may take a list of strings as input, corresponding to the names of input columns from which output column names can be generated. By default input features are named x0, x1, ...

get_n_splits On a *CV splitter* (not an estimator), returns the number of elements one would get if iterating through the return value of *split* given the same parameters. Takes the same parameters as split.

get_params Gets all *parameters*, and their values, that can be set using *set_params*. A parameter `deep` can be used, when set to False to only return those parameters not including __, i.e. not due to indirection via contained estimators.

Most estimators adopt the definition from *base.BaseEstimator*, which simply adopts the parameters defined for `__init__`. *pipeline.Pipeline*, among others, reimplements `get_params` to declare the estimators named in its `steps` parameters as themselves being parameters.

partial_fit Facilitates fitting an estimator in an online fashion. Unlike `fit`, repeatedly calling `partial_fit` does not clear the model, but updates it with respect to the data provided. The portion of data provided to `partial_fit` may be called a mini-batch. Each mini-batch must be of consistent shape, etc. In iterative estimators, `partial_fit` often only performs a single iteration.

`partial_fit` may also be used for *out-of-core* learning, although usually limited to the case where learning can be performed online, i.e. the model is usable after each `partial_fit` and there is no separate processing needed to finalize the model. *cluster.Birch* introduces the convention that calling `partial_fit(X)` will produce a model that is not finalized, but the model can be finalized by calling `partial_fit()` i.e. without passing a further mini-batch.

Generally, estimator parameters should not be modified between calls to `partial_fit`, although `partial_fit` should validate them as well as the new mini-batch of data. In contrast, `warm_start` is used to repeatedly fit the same estimator with the same data but varying parameters.

Like `fit`, `partial_fit` should return the estimator object.

To clear the model, a new estimator should be constructed, for instance with *base.clone*.

predict Makes a prediction for each sample, usually only taking *X* as input (but see under regressor output conventions below). In a *classifier* or *regressor*, this prediction is in the same target space used in fitting (e.g. one of {'red', 'amber', 'green'} if the `y` in fitting consisted of these strings). Despite this, even when `y` passed to *fit* is a list or other array-like, the output of `predict` should always be an array or sparse matrix. In a *clusterer* or *outlier detector* the prediction is an integer.

If the estimator was not already *fitted*, calling this method should raise a *exceptions.NotFittedError*.

Output conventions:

classifier An array of shape `(n_samples,)` `(n_samples, n_outputs)`. *Multilabel* data may be represented as a sparse matrix if a sparse matrix was used in fitting. Each element should be one of the values in the classifier's *classes_* attribute.

clusterer An array of shape `(n_samples,)` where each value is from 0 to `n_clusters - 1` if the corresponding sample is clustered, and -1 if the sample is not clustered, as in *cluster.dbscan*.

outlier detector An array of shape `(n_samples,)` where each value is -1 for an outlier and 1 otherwise.

regressor A numeric array of shape `(n_samples,)`, usually float64. Some regressors have extra options in their `predict` method, allowing them to return standard deviation (`return_std=True`) or covariance (`return_cov=True`) relative to the predicted value. In this case, the return value is a tuple of arrays corresponding to (prediction mean, std, cov) as required.

predict_log_proba The natural logarithm of the output of *predict_proba*, provided to facilitate numerical stability.

predict_proba A method in *classifiers* and *clusterers* that are able to return probability estimates for each class/cluster. Its input is usually only some observed data, *X*.

If the estimator was not already *fitted*, calling this method should raise a *exceptions.NotFittedError*.

Output conventions are like those for *decision_function* except in the *binary* classification case, where one column is output for each class (while `decision_function` outputs a 1d array). For binary and multiclass predictions, each row should add to 1.

Like other methods, `predict_proba` should only be present when the estimator can make probabilistic predictions (see *duck typing*). This means that the presence of the method may depend on estimator parameters (e.g. in *linear_model.SGDClassifier*) or training data (e.g. in *model_selection.GridSearchCV*) and may only appear after fitting.

score A method on an estimator, usually a *predictor*, which evaluates its predictions on a given dataset, and returns a single numerical score. A greater return value should indicate better predictions; accuracy is used for classifiers and R^2 for regressors by default.

If the estimator was not already *fitted*, calling this method should raise a *exceptions.NotFittedError*.

Some estimators implement a custom, estimator-specific score function, often the likelihood of the data under the model.

score_samples TODO

If the estimator was not already *fitted*, calling this method should raise a *exceptions.NotFittedError*.

set_params Available in any estimator, takes keyword arguments corresponding to keys in *get_params*. Each is provided a new value to assign such that calling `get_params` after `set_params` will reflect the changed *parameters*. Most estimators use the implementation in *base.BaseEstimator*, which handles nested parameters and otherwise sets the parameter as an attribute on the estimator. The method is overridden in *pipeline.Pipeline* and related estimators.

split On a *CV splitter* (not an estimator), this method accepts parameters (*X*, *y*, *groups*), where all may be optional, and returns an iterator over `(train_idx, test_idx)` pairs. Each of {train,test}_idx is a 1d integer array, with values from 0 from `X.shape[0] - 1` of any length, such that no values appear in both some `train_idx` and its corresponding `test_idx`.

transform In a *transformer*, transforms the input, usually only *X*, into some transformed space (conventionally notated as *Xt*). Output is an array or sparse matrix of length *n_samples* and with number of columns fixed after *fitting*.

If the estimator was not already *fitted*, calling this method should raise a *exceptions.NotFittedError*.

4.5 Parameters

These common parameter names, specifically used in estimator construction (see concept *parameter*), sometimes also appear as parameters of functions or non-estimator constructors.

class_weight Used to specify sample weights when fitting classifiers as a function of the *target* class. Where *sample_weight* is also supported and given, it is multiplied by the `class_weight` contribution. Similarly,

where `class_weight` is used in a *multioutput* (including *multilabel*) tasks, the weights are multiplied across outputs (i.e. columns of `y`).

By default all samples have equal weight such that classes are effectively weighted by their their prevalence in the training data. This could be achieved explicitly with `class_weight={label1: 1, label2: 1, ...}` for all class labels.

More generally, `class_weight` is specified as a dict mapping class labels to weights (`{class_label: weight}`), such that each sample of the named class is given that weight.

`class_weight='balanced'` can be used to give all classes equal weight by giving each sample a weight inversely related to its class's prevalence in the training data: `n_samples / (n_classes * np.bincount(y))`. Class weights will be used differently depending on the algorithm: for linear models (such as linear SVM or logistic regression), the class weights will alter the loss function by weighting the loss of each sample by its class weight. For tree-based algorithms, the class weights will be used for reweighting the splitting criterion. **Note** however that this rebalancing does not take the weight of samples in each class into account.

For multioutput classification, a list of dicts is used to specify weights for each output. For example, for four-class multilabel classification weights should be `[{0: 1, 1: 1}, {0: 1, 1: 5}, {0: 1, 1: 1}, {0: 1, 1: 1}]` instead of `[{1:1}, {2:5}, {3:1}, {4:1}]`.

The `class_weight` parameter is validated and interpreted with `utils.compute_class_weight`.

cv Determines a cross validation splitting strategy, as used in cross-validation based routines. `cv` is also available in estimators such as *`multioutput.ClassifierChain`* or *`calibration.CalibratedClassifierCV`* which use the predictions of one estimator as training data for another, to not overfit the training supervision.

Possible inputs for `cv` are usually:

- An integer, specifying the number of folds in K-fold cross validation. K-fold will be stratified over classes if the estimator is a classifier (determined by *`base.is_classifier`*) and the *targets* may represent a binary or multiclass (but not multioutput) classification problem (determined by *`utils.multiclass.type_of_target`*).
- A *cross-validation splitter* instance. Refer to the *User Guide* for splitters available within Scikit-learn.
- An iterable yielding train/test splits.

With some exceptions (especially where not using cross validation at all is an option), the default is 3-fold and will change to 5-fold in version 0.22.

`cv` values are validated and interpreted with `utils.check_cv`.

kernel TODO

max_iter For estimators involving iterative optimization, this determines the maximum number of iterations to be performed in *fit*. If `max_iter` iterations are run without convergence, a *`exceptions.ConvergenceWarning`* should be raised. Note that the interpretation of "a single iteration" is inconsistent across estimators: some, but not all, use it to mean a single epoch (i.e. a pass over every sample in the data).

FIXME perhaps we should have some common tests about the relationship between ConvergenceWarning and max_iter.

memory Some estimators make use of `joblib.Memory` to store partial solutions during fitting. Thus when `fit` is called again, those partial solutions have been memoized and can be reused.

A `memory` parameter can be specified as a string with a path to a directory, or a `joblib.Memory` instance (or an object with a similar interface, i.e. a `cache` method) can be used.

`memory` values are validated and interpreted with *`utils.validation.check_memory`*.

metric As a parameter, this is the scheme for determining the distance between two data points. See *metrics.pairwise_distances*. In practice, for some algorithms, an improper distance metric (one that does not obey the triangle inequality, such as Cosine Distance) may be used.

XXX: hierarchical clustering uses `affinity` with this meaning.

We also use *metric* to refer to *evaluation metrics*, but avoid using this sense as a parameter name.

n_components The number of features which a *transformer* should transform the input into. See *components_* for the special case of affine projection.

n_iter_no_change Number of iterations with no improvement to wait before stopping the iterative procedure. This is also known as a *patience* parameter. It is typically used with *early stopping* to avoid stopping too early.

n_jobs This is used to specify how many concurrent processes/threads should be used for parallelized routines. Scikit-learn uses one processor for its processing by default, although it also makes use of NumPy, which may be configured to use a threaded numerical processor library (like MKL; see *FAQ*).

`n_jobs` is an int, specifying the maximum number of concurrently running jobs. If set to -1, all CPUs are used. If 1 is given, no joblib level parallelism is used at all, which is useful for debugging. Even with `n_jobs = 1`, parallelism may occur due to numerical processing libraries (see *FAQ*). For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for `n_jobs = -2`, all CPUs but one are used.

`n_jobs=None` means *unset*; it will generally be interpreted as `n_jobs=1`, unless the current `joblib.Parallel` backend context specifies otherwise.

The use of `n_jobs`-based parallelism in estimators varies:

- Most often parallelism happens in *fitting*, but sometimes parallelism happens in prediction (e.g. in random forests).
- Some parallelism uses a multi-threading backend by default, some a multi-processing backend. It is possible to override the default backend by using *sklearn.utils.parallel_backend*.
- Whether parallel processing is helpful at improving runtime depends on many factors, and it's usually a good idea to experiment rather than assuming that increasing the number of jobs is always a good thing. *It can be highly detrimental to performance to run multiple copies of some estimators or functions in parallel.*

Nested uses of `n_jobs`-based parallelism with the same backend will result in an exception. So `GridSearchCV(OneVsRestClassifier(SVC(), n_jobs=2), n_jobs=2)` won't work.

When `n_jobs` is not 1, the estimator being parallelized must be picklable. This means, for instance, that lambdas cannot be used as estimator parameters.

random_state Whenever randomization is part of a Scikit-learn algorithm, a `random_state` parameter may be provided to control the random number generator used. Note that the mere presence of `random_state` doesn't mean that randomization is always used, as it may be dependent on another parameter, e.g. `shuffle`, being set.

`random_state`'s value may be:

None (default) Use the global random state from `numpy.random`.

An integer Use a new random number generator seeded by the given integer. To make a randomized algorithm deterministic (i.e. running it multiple times will produce the same result), an arbitrary integer `random_state` can be used. However, it may be worthwhile checking that your results are stable across a number of different distinct random seeds. Popular integer random seeds are 0 and 42.

A `numpy.random.RandomState` instance Use the provided random state, only affecting other users of the same random state instance. Calling fit multiple times will reuse the same instance, and will produce different results.

utils.check_random_state is used internally to validate the input `random_state` and return a `RandomState` instance.

scoring Specifies the score function to be maximized (usually by *cross validation*), or – in some cases – multiple score functions to be reported. The score function can be a string accepted by *metrics.get_scorer* or a callable *scorer*, not to be confused with an *evaluation metric*, as the latter have a more diverse API. `scoring` may also be set to None, in which case the estimator's *score* method is used. See *The scoring parameter: defining model evaluation rules* in the User Guide.

Where multiple metrics can be evaluated, `scoring` may be given either as a list of unique strings or a dict with names as keys and callables as values. Note that this does *not* specify which score function is to be maximised, and another parameter such as `refit` may be used for this purpose.

The `scoring` parameter is validated and interpreted using *metrics.check_scoring*.

verbose Logging is not handled very consistently in Scikit-learn at present, but when it is provided as an option, the `verbose` parameter is usually available to choose no logging (set to False). Any True value should enable some logging, but larger integers (e.g. above 10) may be needed for full verbosity. Verbose logs are usually printed to Standard Output. Estimators should not produce any output on Standard Output with the default `verbose` setting.

warm_start When fitting an estimator repeatedly on the same dataset, but for multiple parameter values (such as to find the value maximizing performance as in *grid search*), it may be possible to reuse aspects of the model learnt from the previous parameter value, saving time. When `warm_start` is true, the existing *fitted* model *attributes* are used to initialise the new model in a subsequent call to *fit*.

Note that this is only applicable for some models and some parameters, and even some orders of parameter values. For example, `warm_start` may be used when building random forests to add more trees to the forest (increasing `n_estimators`) but not to reduce their number.

partial_fit also retains the model between calls, but differs: with `warm_start` the parameters change and the data is (more-or-less) constant across calls to `fit`; with `partial_fit`, the mini-batch of data changes and model parameters stay fixed.

There are cases where you want to use `warm_start` to fit on different, but closely related data. For example, one may initially fit to a subset of the data, then fine-tune the parameter search on the full dataset. For classification, all data in a sequence of `warm_start` calls to `fit` must include samples from each class.

4.6 Attributes

See concept *attribute*.

classes_ A list of class labels known to the *classifier*, mapping each label to a numerical index used in the model representation our output. For instance, the array output from *predict_proba* has columns aligned with `classes_`. For *multi-output* classifiers, `classes_` should be a list of lists, with one class listing for each output. For each output, the classes should be sorted (numerically, or lexicographically for strings).

`classes_` and the mapping to indices is often managed with *preprocessing.LabelEncoder*.

components_ An affine transformation matrix of shape `(n_components, n_features)` used in many linear *transformers* where *n_components* is the number of output features and *n_features* is the number of input features.

See also *components_* which is a similar attribute for linear predictors.

coef_ The weight/coefficient matrix of a generalised linear model *predictor*, of shape `(n_features,)` for binary classification and single-output regression, `(n_classes, n_features)` for multiclass classification and `(n_targets, n_features)` for multi-output regression. Note this does not include the intercept (or bias) term, which is stored in `intercept_`.

When available, `feature_importances_` is not usually provided as well, but can be calculated as the norm of each feature's entry in `coef_`.

See also *components_* which is a similar attribute for linear transformers.

embedding_ An embedding of the training data in *manifold learning* estimators, with shape `(n_samples, n_components)`, identical to the output of *fit_transform*. See also *labels_*.

n_iter_ The number of iterations actually performed when fitting an iterative estimator that may stop upon convergence. See also *max_iter*.

feature_importances_ A vector of shape `(n_features,)` available in some *predictors* to provide a relative measure of the importance of each feature in the predictions of the model.

labels_ A vector containing a cluster label for each sample of the training data in *clusterers*, identical to the output of *fit_predict*. See also *embedding_*.

4.7 Data and sample properties

See concept *sample property*.

groups Used in cross validation routines to identify samples which are correlated. Each value is an identifier such that, in a supporting *CV splitter*, samples from some `groups` value may not appear in both a training set and its corresponding test set. See *Cross-validation iterators for grouped data.*.

sample_weight A relative weight for each sample. Intuitively, if all weights are integers, a weighted model or score should be equivalent to that calculated when repeating the sample the number of times specified in the weight. Weights may be specified as floats, so that sample weights are usually equivalent up to a constant positive scaling factor.

FIXME Is this interpretation always the case in practice? We have no common tests.

Some estimators, such as decision trees, support negative weights. FIXME: This feature or its absence may not be tested or documented in many estimators.

This is not entirely the case where other parameters of the model consider the number of samples in a region, as with `min_samples` in *`cluster.DBSCAN`*. In this case, a count of samples becomes to a sum of their weights.

In classification, sample weights can also be specified as a function of class with the *class_weight* estimator *parameter*.

X Denotes data that is observed at training and prediction time, used as independent variables in learning. The notation is uppercase to denote that it is ordinarily a matrix (see *rectangular*). When a matrix, each sample may be represented by a *feature* vector, or a vector of *precomputed* (dis)similarity with each training sample. `X` may also not be a matrix, and may require a *feature extractor* or a *pairwise metric* to turn it into one before learning a model.

Xt Shorthand for "transformed *X*".

y

Y Denotes data that may be observed at training time as the dependent variable in learning, but which is unavailable at prediction time, and is usually the *target* of prediction. The notation may be uppercase to denote that it is a matrix, representing *multi-output* targets, for instance; but usually we use `y` and sometimes do so even when multiple outputs are assumed.