Glossary of Common Terms and API Elements

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 1)

Unknown directive type "currentmodule".

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

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: :ref:`glossary_estimator_types`, :ref:`glossary_target_types`, :ref:`glossary_methods`, :ref:`glossary_parameters`, :ref:`glossary_attributes`, :ref:`glossary_sample_props`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 16); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 16); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 16); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 16); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 16); *backlink*

Unknown interpreted text role "ref".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 16); *backlink*

Unknown interpreted text role "ref".

General Concepts

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main) (doc)glossary.rst`, line 25)

Unknown directive type "glossary".

```
.. glossary::

    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 :ref:`overviewed in the
        contributor documentation <api_overview>`.

        The specific interfaces that constitute Scikit-learn's public API are
        largely documented in :ref:`api_ref`. 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 :term:`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
    :func:`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 :class:`pandas.DataFrame` with all columns numeric
    * a numeric :class:`pandas.Series`

    It excludes:

    * a :term:`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 :class:`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 :term:`fit` or
    :term:`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; :term:`transductive` outputs such as :term:`labels_` or
    :term:`embedding_`; or diagnostic data, such as
    :term:`feature_importances_`.
    Common attributes are listed :ref:`below <glossary_attributes>`.

    A public attribute may have the same name as a constructor
    :term:`parameter`, with a ``_`` appended.  This is used to store a
    validated or estimated version of the user's input. For example,
    :class:`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__``
    :term:`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 backward 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 behavior 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 :term:`deprecation` period
        (usually two releases long).  Warnings are issued using Python's
        :mod:`warnings` module.
    Keyword arguments
        We may sometimes assume that all optional parameters (other than X
        and y to :term:`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 :term:`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 :ref:`persistence_limitations`.
    :func:`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 license.  When a release inadvertently
    introduces changes that are not backward 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()
    <https://docs.python.org/3/library/functions.html#callable>`_.

categorical feature
    A categorical or nominal :term:`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.
    :class:`~sklearn.preprocessing.OrdinalEncoder` helps encoding
    string-valued categorical features as ordinal integers, and
    :class:`~sklearn.preprocessing.OneHotEncoder` can be used to
    one-hot encode categorical features.
    See also :ref:`preprocessing_categorical_features` and the
    `categorical-encoding
    <https://github.com/scikit-learn-contrib/category_encoders>`_
    package for tools related to encoding categorical features.

clone
cloned
    To copy an :term:`estimator instance` and create a new one with
    identical :term:`parameters`, but without any fitted
    :term:`attributes`, using :func:`~sklearn.base.clone`.

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

    If the estimator's `random_state` parameter is an integer (or if the
    estimator doesn't have a `random_state` parameter), an *exact clone*
    is returned: the clone and the original estimator will give the exact
    same results. Otherwise, *statistical clone* is returned: the clone
    might yield different results from the original estimator. More
    details can be found in :ref:`randomness`.

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
    :func:`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 :term:`estimator
    tags`.

deprecation
    We use deprecation to slowly violate our :term:`backwards
    compatibility` assurances, usually 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 :ref:`Contributors' Guide <contributing_deprecation>`.

dimensionality
    May be used to refer to the number of :term:`features` (i.e.
    :term:`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
    <https://www.python.org/dev/peps/pep-0257/>`_, and follow `NumpyDoc
    conventions <https://numpydoc.readthedocs.io/en/latest/format.html>`_.

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
    :term:`set_params` and hence in specifying a search grid in
    :ref:`parameter search <grid_search>`. See :term:`parameter`.
    It is also used in :meth:`pipeline.Pipeline.fit` for passing
    :term:`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.

    Our documentation can sometimes give information about the dtype
    precision, e.g. `np.int32`, `np.int64`, etc. When the precision is
    provided, it refers to the NumPy dtype. If an arbitrary precision is
    used, the documentation will refer to dtype `integer` or `floating`.
    Note that in this case, the precision can be platform dependent.
    The `numeric` dtype refers to accepting both `integer` and `floating`.

    TODO: Mention efficiency and precision issues; casting policy.

duck typing
    We try to apply `duck typing
    <https://en.wikipedia.org/wiki/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
      :term:`fitted`.  For instance, we cannot a priori determine if
      :term:`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 :term:`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
      :term:`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 :term:`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 :term:`n_iter_no_change`.

estimator instance
    We sometimes use this terminology to distinguish an :term:`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 :ref:`example gallery <general_examples>`
      rendered (using `sphinx-gallery
      <https://sphinx-gallery.readthedocs.io/>`_) 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 :ref:`User Guide <user_guide>` (built from ``doc/``)
      alongside a technical description of the estimator.

experimental
    An experimental tool is already usable but its public API, such as
    default parameter values or fitted attributes, is still subject to
    change in future versions without the usual :term:`deprecation`
    warning policy.

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 :mod:`metrics`
    (disregarding :mod:`metrics.pairwise`), as distinct from the
    :term:`score` method and the :term:`scoring` API used in cross
    validation. See :ref:`model_evaluation`.

    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 :term:`predict` (``y_pred``),
    of :term:`predict_proba` (``y_proba``), or of an arbitrary score
    function including :term:`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 :mod:`metrics` and are estimator-specific, notably model
    likelihoods.

estimator tags
    A proposed feature (e.g. :issue:`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 :term:`common tests`.

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

    .. glossary::

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

    For more detailed info, see :ref:`estimator_tags`.

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 :term:`categorical feature` and
    :term:`missing values`.

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

fitting
    Calling :term:`fit` (or :term:`fit_transform`, :term:`fit_predict`,
    etc.) on an estimator.

fitted
    The state of an estimator after :term:`fitting`.

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

    * should raise :class:`exceptions.NotFittedError` when a prediction
      method (:term:`predict`, :term:`transform`, etc.) is called.
      (:func:`utils.validation.check_is_fitted` is used internally
      for this purpose.)
    * should not have any :term:`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
    :term:`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
    :func:`linear_model.enet_path`.  For transductive models, this also
    returns the embedding or cluster labels, as in
    :func:`manifold.spectral_embedding` or :func:`cluster.dbscan`.  Many
    preprocessing transformers also provide a function interface, akin to
    calling :term:`fit_transform`, as in
    :func:`preprocessing.maxabs_scale`.  Users should be careful to avoid
    :term:`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 :term:`examples`.

hyperparameter
hyper-parameter
    See :term:`parameter`.

impute
imputation
    Most machine learning algorithms require that their inputs have no
    :term:`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 :term:`array-like`, :term:`sparse matrix`, pandas DataFrame or
    sequence (usually a list).

induction
inductive
    Inductive (contrasted with :term:`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 :term:`predict`
    and/or :term:`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
    :term:`memory mapping`. See :ref:`parallelism` 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 :term:`transformer` to the entirety of a dataset rather
    than each training portion in a cross validation split.

    We aim to provide interfaces (such as :mod:`pipeline` and
    :mod:`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 :obj:`numpy.memmap`. When
    using :term:`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 :class:`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 :term:`imputation` or
    learning can be performed in integer space.
    :term:`Unlabeled data <unlabeled data>` is a special case of missing
    values in the :term:`target`.

``n_features``
    The number of :term:`features`.

``n_outputs``
    The number of :term:`outputs` in the :term:`target`.

``n_samples``
    The number of :term:`samples`.

``n_targets``
    Synonym for :term:`n_outputs`.

narrative docs
narrative documentation
    An alias for :ref:`User Guide <user_guide>`, i.e. documentation written
    in ``doc/modules/``. Unlike the :ref:`API reference <api_ref>` 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 :term:`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 :term:`targets` soon after making predictions on corresponding
    batch of data.  Intrinsically, the model must be usable for prediction
    after each batch. See :term:`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
    :term:`partial_fit`.

outputs
    Individual scalar/categorical variables per sample in the
    :term:`target`.  For example, in multilabel classification each
    possible label corresponds to a binary output. Also called *responses*,
    *tasks* or *targets*.
    See :term:`multiclass multioutput` and :term:`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 :class:`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 :term:`n_jobs` for controlling parallelism.

    When talking about the parameters of a :term:`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 :term:`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 :term:`estimator instance` using its :term:`get_params` method.

    Between construction and fitting, parameters may be modified using
    :term:`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 :term:`fit` is
    called.

    Common parameters are listed :ref:`below <glossary_parameters>`.

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 :term:`feature vector`).  We particularly provide
    implementations of distance metrics (as well as improper metrics like
    Cosine Distance) through :func:`metrics.pairwise_distances`, and of
    kernel functions (a constrained class of similarity functions) in
    :func:`metrics.pairwise_kernels`.  These can compute pairwise distance
    matrices that are symmetric and hence store data redundantly.

    See also :term:`precomputed` and :term:`metric`.

    Note that for most distance metrics, we rely on implementations from
    :mod:`scipy.spatial.distance`, but may reimplement for efficiency in
    our context. The :class:`metrics.DistanceMetric` interface is used to implement
    distance metrics for integration with efficient neighbors search.

pd
    A shorthand for `Pandas <https://pandas.pydata.org>`_ due to the
    conventional import statement::

        import pandas as pd

precomputed
```

```
    Where algorithms rely on :term:`pairwise metrics`, and can be computed
    from pairwise metrics alone, we often allow the user to specify that
    the :term:`X` provided is already in the pairwise (dis)similarity
    space, rather than in a feature space.  That is, when passed to
    :term:`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'. If
    this is the case, then the estimator should set the `pairwise`
    estimator tag as True.

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

    This term excludes samples with non-vectorial structures, such as text,
    an image of arbitrary size, a time series of arbitrary length, a set of
    vectors, etc. The purpose of a :term:`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 :term:`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 :term:`features` (``X``) and
    :term:`target` (``y``). The most prominent example is
    :term:`sample_weight`; see others at :ref:`glossary_sample_props`.

    As of version 0.19 we do not have a consistent approach to handling
    sample properties and their routing in :term:`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 :ref:`SLEP <slep>` and follows the
    decision-making process outlined in :ref:`governance`.
    For all votes, a proposal must have been made public and discussed before the
    vote. Such a proposal must be a consolidated document, in the form of a
    â€˜Scikit-Learn Enhancement Proposalâ€™ (SLEP), rather than a long discussion on an
    issue. A SLEP must be submitted as a pull-request to
    `enhancement proposals <https://scikit-learn-enhancement-proposals.readthedocs.io>`_ using the
    `SLEP template <https://scikit-learn-enhancement-proposals.readthedocs.io/en/latest/slep_template.htr

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
    :term:`fitting` the model.  We conventionally apply the label ``-1``
    to :term:`unlabeled` samples in semi-supervised classification.

sparse matrix
sparse graph
    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 :mod:`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 :term:`multilabel indicator
        matrices`.
    graph semantics
        As with :mod:`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. :func:`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 :term:`fitting` the model, provided as
        :term:`y`.  This is the approach taken in a :term:`classifier` or
        :term:`regressor` among other estimators.

    target
    targets
        The *dependent variable* in :term:`supervised` (and
        :term:`semisupervised`) learning, passed as :term:`y` to an estimator's
        :term:`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 :ref:`glossary_target_types`.

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

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

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

Class APIs and Estimator Types

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 794)

Unknown directive type "glossary".

```
.. glossary::

    classifier
    classifiers
        A :term:`supervised` (or :term:`semi-supervised`) :term:`predictor`
        with a finite set of discrete possible output values.

        A classifier supports modeling some of :term:`binary`,
        :term:`multiclass`, :term:`multilabel`, or :term:`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 :term:`classes_` attribute after fitting,
        and usually inherit from :class:`base.ClassifierMixin`, which sets
        their :term:`_estimator_type` attribute.

        A classifier can be distinguished from other estimators with
        :func:`~base.is_classifier`.

        A classifier must implement:

        * :term:`fit`
        * :term:`predict`
        * :term:`score`

        It may also be appropriate to implement :term:`decision_function`,
        :term:`predict_proba` and :term:`predict_log_proba`.

    clusterer
    clusterers
        A :term:`unsupervised` :term:`predictor` with a finite set of discrete
        output values.

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

```
    A clusterer must implement:

    * :term:`fit`
    * :term:`fit_predict` if :term:`transductive`
    * :term:`predict` if :term:`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:

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

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

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

    The core functionality of some estimators may also be available as a
    :term:`function`.

feature extractor
feature extractors
    A :term:`transformer` which takes input where each sample is not
    represented as an :term:`array-like` object of fixed length, and
    produces an :term:`array-like` object of :term:`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 :term:`rectangular` data.

    Feature extractors must implement at least:

    * :term:`fit`
    * :term:`transform`
    * :term:`get_feature_names`
    * :term:`get_feature_names_out`

meta-estimator
meta-estimators
metaestimator
metaestimators
    An :term:`estimator` which takes another estimator as a parameter.
    Examples include :class:`pipeline.Pipeline`,
    :class:`model_selection.GridSearchCV`,
    :class:`feature_selection.SelectFromModel` and
    :class:`ensemble.BaggingClassifier`.

    In a meta-estimator's :term:`fit` method, any contained estimators
    should be :term:`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 pre-fitted
    estimator (e.g. using ``prefit=True`` in
    :class:`feature_selection.SelectFromModel`). One known issue with this
    is that the pre-fitted 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 pre-fitted.

    In cases where a meta-estimator's primary behaviors (e.g.
    :term:`predict` or :term:`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 :term:`fitted` (see also
    :term:`duck typing`), for which
    :func:`utils.metaestimators.available_if` may help.  It
    should also provide (or modify) the :term:`estimator tags` and
    :term:`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 :term:`rectangular`.

outlier detector
outlier detectors
    An :term:`unsupervised` binary :term:`predictor` which models the
    distinction between core and outlying samples.

    Outlier detectors must implement:

    * :term:`fit`
    * :term:`fit_predict` if :term:`transductive`
    * :term:`predict` if :term:`inductive`
```

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

    predictor
    predictors
        An :term:`estimator` supporting :term:`predict` and/or
        :term:`fit_predict`. This encompasses :term:`classifier`,
        :term:`regressor`, :term:`outlier detector` and :term:`clusterer`.

        In statistics, "predictors" refers to :term:`features`.

    regressor
    regressors
        A :term:`supervised` (or :term:`semi-supervised`) :term:`predictor`
        with :term:`continuous` output values.

        Regressors usually inherit from :class:`base.RegressorMixin`, which
        sets their :term:`_estimator_type` attribute.

        A regressor can be distinguished from other estimators with
        :func:`~base.is_regressor`.

        A regressor must implement:

        * :term:`fit`
        * :term:`predict`
        * :term:`score`

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

    vectorizer
    vectorizers
        See :term:`feature extractor`.
```

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

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst`, line 972)

Unknown directive type "glossary".

```
.. glossary::

    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 :ref:`cross_validation`),
        by providing :term:`split` and :term:`get_n_splits` methods.
        Note that unlike estimators, these do not have :term:`fit` methods
        and do not provide :term:`set_params` or :term:`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 :ref:`User
        Guide <grid_search>`). Some example of cross-validation estimators
        are :class:`ElasticNetCV <linear_model.ElasticNetCV>` and
        :class:`LogisticRegressionCV <linear_model.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
        :term:`estimator` class along with :ref:`grid search <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
        :class:`RidgeCV <linear_model.RidgeCV>` class, which can instead
        perform efficient Leave-One-Out (LOO) CV. By default, all these
        estimators, apart from :class:`RidgeCV <linear_model.RidgeCV>` with an
        LOO-CV, will be refitted on the full training dataset after finding the
        best combination of hyper-parameters.

    scorer
        A non-estimator callable object which evaluates an estimator on given
        test data, returning a number. Unlike :term:`evaluation metrics`,
        a greater returned number must correspond with a *better* score.
        See :ref:`scoring_parameter`.
```

Further examples:

- :class:`metrics.DistanceMetric`

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst`, line 1011); *backlink*

Unknown interpreted text role "class".

- :class:`gaussian_process.kernels.Kernel`

  System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1012); *backlink*

  Unknown interpreted text role "class".

- `tree.Criterion`

Target Types

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1020)

Unknown directive type "glossary".

```
.. glossary::

    binary
        A classification problem consisting of two classes.  A binary target
        may  be represented as for a :term:`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 :term:`pos_label` in
        :term:`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
        :term:`decision_function`, for instance.

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

        :func:`~utils.multiclass.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).

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

    continuous multioutput
    continuous multi-output
    multioutput continuous
    multi-output continuous
        A regression problem where each sample's target consists of ``n_outputs``
        :term:`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
        :term:`continuous` targets, horizontally stacked into an array
        of shape ``(n_samples, n_outputs)``.

        :func:`~utils.multiclass.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
    multi-class
        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 :term:`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, :term:`unlabeled` samples should
        have the special label -1 in ``y``.

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

        A :class:`preprocessing.LabelEncoder` helps to canonicalize multiclass
        targets as integers.

        :func:`~utils.multiclass.type_of_target` will return 'multiclass' for
        multiclass input. The user may also want to handle 'binary' input
        identically to 'multiclass'.

    multiclass multioutput
    multi-class multi-output
    multioutput multiclass
    multi-output multi-class
        A classification problem where each sample's target consists of
```

```
        ``n_outputs`` :term:`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 labeled 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
        :term:`multilabel`.

        Multiclass multioutput targets are represented as multiple
        :term:`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.

        :mod:`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.

        :func:`~utils.multiclass.type_of_target` will return
        'multiclass-multioutput' for multiclass multioutput input.

    multilabel
    multi-label
        A :term:`multiclass multioutput` target where each output is
        :term:`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,
        :class:`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
        :class:`preprocessing.LabelBinarizer` turns it into a multilabel
        problem.

        :func:`~utils.multiclass.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 :term:`multiclass multioutput` and :term:`continuous
        multioutput`. We do not currently support modelling mixed
        classification and regression targets.
```

Methods

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1152)

Unknown directive type "glossary".

```
.. glossary::

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

        If the estimator was not already :term:`fitted`, calling this method
        should raise a :class:`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
            :term:`classes_`).
        multiclass classification
            A 2-dimensional array, where the row-wise arg-maximum is the
            predicted class.  Columns are ordered according to
            :term:`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
            <https://gist.github.com/jnothman/4807b1b0266613c20ba4d1f88d0f8cf5>`_
            highlights 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
        :term:`samples` ``X``, :term:`targets` ``y`` if the model is supervised,
        and potentially other :term:`sample properties` such as
        :term:`sample_weight`.  It should:

        * clear any prior :term:`attributes` stored on the estimator, unless
          :term:`warm_start` is used;
        * validate and interpret any :term:`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 :term:`fitted` estimator to facilitate method
          chaining.

        :ref:`glossary_target_types` describes possible formats for ``y``.

    ``fit_predict``
        Used especially for :term:`unsupervised`, :term:`transductive`
        estimators, this fits the model and returns the predictions (similar to
        :term:`predict`) on the training data. In clusterers, these predictions
        are also stored in the :term:`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 :term:`transformers` which fits the estimator and returns
        the transformed training data. It takes parameters as in :term:`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).
        :term:`Transductive <transductive>` transformers may also provide
        ``fit_transform`` but not :term:`transform`.

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

        In :term:`inductive` learning -- where the goal is to learn a
        generalized 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 :term:`data leakage`.

    ``get_feature_names``
        Primarily for :term:`feature extractors`, but also used for other
        transformers to provide string names for each column in the output of
        the estimator's :term:`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_feature_names_out``
        Primarily for :term:`feature extractors`, but also used for other
        transformers to provide string names for each column in the output of
        the estimator's :term:`transform` method.  It outputs an array of
        strings and may take an array-like of strings as input, corresponding
        to the names of input columns from which output column names can
        be generated.  If `input_features` is not passed in, then the
        `feature_names_in_` attribute will be used. If the
        `feature_names_in_` attribute is not defined, then the
        input names are named `[x0, x1, ..., x(n_features_in_ - 1)]`.

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

    ``get_params``
        Gets all :term:`parameters`, and their values, that can be set using
        :term:`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 :class:`base.BaseEstimator`,
        which simply adopts the parameters defined for ``__init__``.
        :class:`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 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 :term:`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.
        :class:`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 :func:`base.clone`.

        NOTE: Using ``partial_fit`` after ``fit`` results in undefined behavior.

    ``predict``
        Makes a prediction for each sample, usually only taking :term:`X` as
        input (but see under regressor output conventions below). In a
        :term:`classifier` or :term:`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 :term:`fit` is a list or other array-like, the
        output of ``predict`` should always be an array or sparse matrix. In a
        :term:`clusterer` or :term:`outlier detector` the prediction is an
        integer.

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

        Output conventions:

        classifier
            An array of shape ``(n_samples,)`` ``(n_samples, n_outputs)``.
            :term:`Multilabel <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 :term:`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
            :func:`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 :term:`predict_proba`, provided
        to facilitate numerical stability.

    ``predict_proba``
        A method in :term:`classifiers` and :term:`clusterers` that can
        return probability estimates for each class/cluster.  Its input is
        usually only some observed data, :term:`X`.

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

        Output conventions are like those for :term:`decision_function` except
        in the :term:`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 :term:`duck typing`).
        This means that the presence of the method may depend on estimator
        parameters (e.g. in :class:`linear_model.SGDClassifier`) or training
        data (e.g. in :class:`model_selection.GridSearchCV`) and may only
        appear after fitting.

    ``score``
        A method on an estimator, usually a :term:`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 :term:`fitted`, calling this method
        should raise a :class:`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 :term:`fitted`, calling this method
        should raise a :class:`exceptions.NotFittedError`.

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

    ``split``
        On a :term:`CV splitter` (not an estimator), this method accepts
        parameters (:term:`X`, :term:`y`, :term:`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 :term:`transformer`, transforms the input, usually only :term:`X`,
        into some transformed space (conventionally notated as :term:`Xt`).
        Output is an array or sparse matrix of length :term:`n_samples` and
        with the number of columns fixed after :term:`fitting`.

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

Parameters

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1425); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1429)

Unknown directive type "glossary".

```
.. glossary::

    ``class_weight``
        Used to specify sample weights when fitting classifiers as a function
        of the :term:`target` class.  Where :term:`sample_weight` is also
        supported and given, it is multiplied by the ``class_weight``
        contribution. Similarly, where ``class_weight`` is used in a
        :term:`multioutput` (including :term:`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 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
        :func:`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 :class:`multioutput.ClassifierChain` or
        :class:`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 :func:`base.is_classifier`) and the
          :term:`targets` may represent a binary or multiclass (but not
          multioutput) classification problem (determined by
          :func:`utils.multiclass.type_of_target`).
        - A :term:`cross-validation splitter` instance. Refer to the
          :ref:`User Guide <cross_validation>` 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 5-fold.

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

    ``kernel``
        TODO

    ``max_iter``
        For estimators involving iterative optimization, this determines the
        maximum number of iterations to be performed in :term:`fit`.  If
        ``max_iter`` iterations are run without convergence, a
        :class:`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 :class:`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 :class:`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
        :func:`utils.validation.check_memory`.

    ``metric``
        As a parameter, this is the scheme for determining the distance between
        two data points.  See :func:`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 :term:`evaluation metrics`, but avoid
        using this sense as a parameter name.

    ``n_components``
        The number of features which a :term:`transformer` should transform the
        input into. See :term:`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 :term:`early stopping` to avoid stopping too
        early.

    ``n_jobs``
        This parameter is used to specify how many concurrent processes or
        threads should be used for routines that are parallelized with
        :term:`joblib`.

        ``n_jobs`` is an integer, specifying the maximum number of concurrently
        running workers. If 1 is given, no joblib parallelism is used at all,
        which is useful for debugging. If set to -1, all CPUs are used. For
        ``n_jobs`` below -1, (n_cpus + 1 + n_jobs) are used. For example with
        ``n_jobs=-2``, all CPUs but one are used.

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

        Note that even if ``n_jobs=1``, low-level parallelism (via Numpy and OpenMP)
        might be used in some configuration.

        For more details on the use of ``joblib`` and its interactions with
        scikit-learn, please refer to our :ref:`parallelism notes
        <parallelism>`.

    ``pos_label``
        Value with which positive labels must be encoded in binary
        classification problems in which the positive class is not assumed.
        This value is typically required to compute asymmetric evaluation
        metrics such as precision and recall.

    ``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.

        The passed value will have an effect on the reproducibility of the
        results returned by the function (:term:`fit`, :term:`split`, or any
        other function like :func:`~sklearn.cluster.k_means`). `random_state`'s
        value may be:

        None (default)
            Use the global random state instance from :mod:`numpy.random`.
            Calling the function multiple times will reuse
            the same instance, and will produce different results.

        An integer
            Use a new random number generator seeded by the given integer.
            Using an int will produce the same results across different calls.
            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
            <https://en.wikipedia.org/wiki/Answer_to_the_Ultimate_Question_of_Life%2C_the_Universe%2C_and_Eve
            Integer values must be in the range `[0, 2**32 - 1]`.

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

        :func:`utils.check_random_state` is used internally to validate the
        input ``random_state`` and return a :class:`~numpy.random.RandomState`
        instance.

        For more details on how to control the randomness of scikit-learn
        objects and avoid common pitfalls, you may refer to :ref:`randomness`.

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

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


        The ``scoring`` parameter is validated and interpreted using
        :func:`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 :ref:`grid search <grid_search>`), it may be possible
        to reuse aspects of the model learned from the previous parameter value,
        saving time.  When ``warm_start`` is true, the existing :term:`fitted`
        model :term:`attributes` are used to initialize the new model
        in a subsequent call to :term:`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.

        :term:`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.
```

Attributes

See concept :term:`attribute`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1673); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1675)

Unknown directive type "glossary".

```
.. glossary::

    ``classes_``
        A list of class labels known to the :term:`classifier`, mapping each
        label to a numerical index used in the model representation our output.
        For instance, the array output from :term:`predict_proba` has columns
        aligned with ``classes_``. For :term:`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
        :class:`preprocessing.LabelEncoder`.

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

        See also :term:`components_` which is a similar attribute for linear
        predictors.

    ``coef_``
        The weight/coefficient matrix of a generalised linear model
        :term:`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 :term:`components_` which is a similar attribute for linear
        transformers.

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

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

    ``feature_importances_``
        A vector of shape ``(n_features,)`` available in some
        :term:`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 :term:`clusterers`, identical to the output of
        :term:`fit_predict`.  See also :term:`embedding_`.
```

Data and sample properties

See concept :term:`sample property`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1738); *backlink*

Unknown interpreted text role "term".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\scikit-learn-main\doc\(scikit-learn-main)(doc)glossary.rst, line 1740)

Unknown directive type "glossary".

```
.. glossary::

    ``groups``
        Used in cross-validation routines to identify samples that are correlated.
        Each value is an identifier such that, in a supporting
```

```
        :term:`CV splitter`, samples from some ``groups`` value may not
        appear in both a training set and its corresponding test set.
        See :ref:`group_cv`.

    ``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
        :class:`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 :term:`class_weight` estimator :term:`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 :term:`rectangular`).
        When a matrix, each sample may be represented by a :term:`feature`
        vector, or a vector of :term:`precomputed` (dis)similarity with each
        training sample. ``X`` may also not be a matrix, and may require a
        :term:`feature extractor` or a :term:`pairwise metric` to turn it into
        one before learning a model.

    ``Xt``
        Shorthand for "transformed :term:`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 :term:`target` of prediction.  The notation may be
        uppercase to denote that it is a matrix, representing
        :term:`multi-output` targets, for instance; but usually we use ``y``
        and sometimes do so even when multiple outputs are assumed.
```