Examples

```
>>> pd.read_table('data.csv')
```

pandas.read_csv

`pandas.read_csv`(*filepath_or_buffer*, *sep=NoDefault.no_default*, *delimiter=None*, *header='infer'*, *names=NoDefault.no_default*, *index_col=None*, *usecols=None*, *squeeze=None*, *prefix=NoDefault.no_default*, *mangle_dupe_cols=True*, *dtype=None*, *engine=None*, *converters=None*, *true_values=None*, *false_values=None*, *skipinitialspace=False*, *skiprows=None*, *skipfooter=0*, *nrows=None*, *na_values=None*, *keep_default_na=True*, *na_filter=True*, *verbose=False*, *skip_blank_lines=True*, *parse_dates=None*, *infer_datetime_format=False*, *keep_date_col=False*, *date_parser=None*, *dayfirst=False*, *cache_dates=True*, *iterator=False*, *chunksize=None*, *compression='infer'*, *thousands=None*, *decimal='.'*, *lineterminator=None*, *quotechar='"'*, *quoting=0*, *doublequote=True*, *escapechar=None*, *comment=None*, *encoding=None*, *encoding_errors='strict'*, *dialect=None*, *error_bad_lines=None*, *warn_bad_lines=None*, *on_bad_lines=None*, *delim_whitespace=False*, *low_memory=True*, *memory_map=False*, *float_precision=None*, *storage_options=None*)

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for IO Tools.

Parameters

filepath_or_buffer [str, path object or file-like object] Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via builtin `open` function) or `StringIO`.

sep [str, default ','] Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`.

delimiter [str, default `None`] Alias for sep.

header [int, list of int, None, default 'infer'] Row number(s) to use as the column names, and the start of the data. Default behavior is to infer the column names: if no names are passed the behavior is identical to `header=0` and column names are inferred from the first line of the file, if column names are passed explicitly then the behavior is identical to `header=None`. Explicitly pass `header=0` to be able to replace existing names. The header can be a list of integers that specify row locations for a multi-index on the columns e.g. [0,1,3]. Intervening rows that are not specified will be skipped (e.g. 2 in this example is skipped). Note that this parameter ignores commented lines and empty lines if `skip_blank_lines=True`, so `header=0` denotes the first line of data rather than the first line of the file.

names [array-like, optional] List of column names to use. If the file contains a header row, then you should explicitly pass `header=0` to override the column names. Duplicates in this list are not allowed.

Summarizing data: describe

There is a convenient *describe()* function which computes a variety of summary statistics about a Series or the columns of a DataFrame (excluding NAs of course):

```
In [94]: series = pd.Series(np.random.randn(1000))

In [95]: series[::2] = np.nan

In [96]: series.describe()
Out[96]:
count    500.000000
mean      -0.021292
std        1.015906
min       -2.683763
25%       -0.699070
50%       -0.069718
75%        0.714483
max        3.160915
dtype: float64

In [97]: frame = pd.DataFrame(np.random.randn(1000, 5), columns=["a", "b", "c", "d", "e
↪"])

In [98]: frame.iloc[::2] = np.nan

In [99]: frame.describe()
Out[99]:
                a           b           c           d           e
count  500.000000  500.000000  500.000000  500.000000  500.000000
mean     0.033387    0.030045   -0.043719   -0.051686    0.005979
std      1.017152    0.978743    1.025270    1.015988    1.006695
min     -3.000951   -2.637901   -3.303099   -3.159200   -3.188821
25%     -0.647623   -0.576449   -0.712369   -0.691338   -0.691115
50%      0.047578   -0.021499   -0.023888   -0.032652   -0.025363
75%      0.729907    0.775880    0.618896    0.670047    0.649748
max      2.740139    2.752332    3.004229    2.728702    3.240991
```

You can select specific percentiles to include in the output:

```
In [100]: series.describe(percentiles=[0.05, 0.25, 0.75, 0.95])
Out[100]:
count    500.000000
mean      -0.021292
std        1.015906
min       -2.683763
5%        -1.645423
25%       -0.699070
50%       -0.069718
75%        0.714483
95%        1.711409
max        3.160915
dtype: float64
```

`matplotlib.pyplot.plasma()`

set the default colormap to plasma and apply to current image if any. See help(colormaps) for more information

`matplotlib.pyplot.plot(*args, **kwargs)`

Plot lines and/or markers to the `Axes`. *args* is a variable length argument, allowing for multiple *x*, *y* pairs with an optional format string. For example, each of the following is legal:

```
plot(x, y)        # plot x and y using default line style and color
plot(x, y, 'bo')  # plot x and y using blue circle markers
plot(y)           # plot y using x as index array 0..N-1
plot(y, 'r+')     # ditto, but with red plusses
```

If *x* and/or *y* is 2-dimensional, then the corresponding columns will be plotted.

If used with labeled data, make sure that the color spec is not included as an element in data, as otherwise the last case `plot("v","r", data={"v":..., "r":...)` can be interpreted as the first case which would do `plot(v, r)` using the default line style and color.

If not used with labeled data (i.e., without a data argument), an arbitrary number of *x*, *y*, *fmt* groups can be specified, as in:

```
a.plot(x1, y1, 'g^', x2, y2, 'g-')
```

Return value is a list of lines that were added.

By default, each line is assigned a different style specified by a 'style cycle'. To change this behavior, you can edit the axes.prop_cycle rcParam.

The following format string characters are accepted to control the line style or marker:

character	description
`'-'`	solid line style
`'--'`	dashed line style
`'-.'`	dash-dot line style
`':'`	dotted line style
`'.'`	point marker
`','`	pixel marker
`'o'`	circle marker
`'v'`	triangle_down marker
`'^'`	triangle_up marker
`'<'`	triangle_left marker
`'>'`	triangle_right marker
`'1'`	tri_down marker
`'2'`	tri_up marker
`'3'`	tri_left marker
`'4'`	tri_right marker
`'s'`	square marker
`'p'`	pentagon marker
`'*'`	star marker
`'h'`	hexagon1 marker
`'H'`	hexagon2 marker
`'+'`	plus marker
`'x'`	x marker
`'D'`	diamond marker
`'d'`	thin_diamond marker
`'|'`	vline marker
`'_'`	hline marker

The following color abbreviations are supported:

character	color
‘b’	blue
‘g’	green
‘r’	red
‘c’	cyan
‘m’	magenta
‘y’	yellow
‘k’	black
‘w’	white

In addition, you can specify colors in many weird and wonderful ways, including full names (`'green'`), hex strings (`'#008000'`), RGB or RGBA tuples (`(0,1,0,1)`) or grayscale intensities as a string (`'0.8'`). Of these, the string specifications can be used in place of a `fmt` group, but the tuple forms can be used only as `kwargs`.

Line styles and colors are combined in a single format string, as in `'bo'` for blue circles.

The *kwargs* can be used to set line properties (any property that has a `set_*` method). You can use this to set a line label (for auto legends), linewidth, anitialising, marker face color, etc. Here is an

Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

Other Parameters **kwargs

Optional parameters are properties of the class matplotlib.patches.Polygon.

See also:

axhspan

Examples

Draw a vertical, green, translucent rectangle from x = 1.25 to x = 1.55 that spans the yrange of the axes.

```
>>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)
```

`matplotlib.pyplot.bar`(*left*, *height*, *width=0.8*, *bottom=None*, *hold=None*, *data=None*, ***kwargs*)

Make a bar plot.

Make a bar plot with rectangles bounded by:

`left`, `left` + `width`, `bottom`, `bottom` + `height` (left, right, bottom and top edges)

Parameters left : sequence of scalars

the x coordinates of the left sides of the bars

height : sequence of scalars

the heights of the bars

width : scalar or array-like, optional

the width(s) of the bars default: 0.8

bottom : scalar or array-like, optional

the y coordinate(s) of the bars default: None

color : scalar or array-like, optional

the colors of the bar faces

edgecolor : scalar or array-like, optional

the colors of the bar edges

linewidth : scalar or array-like, optional

width of bar edge(s). If None, use default linewidth; If 0, don't draw edges. default: None

tick_label : string or array-like, optional

the tick labels of the bars default: None

xerr : scalar or array-like, optional

if not None, will be used to generate errorbar(s) on the bar chart default: None

yerr : scalar or array-like, optional

if not None, will be used to generate errorbar(s) on the bar chart default: None

ecolor : scalar or array-like, optional

specifies the color of errorbar(s) default: None

capsize : scalar, optional

determines the length in points of the error bar caps default: None, which will take the value from the `errorbar.capsize` *rcParam*.

error_kw : dict, optional

dictionary of kwargs to be passed to errorbar method. *ecolor* and *capsize* may be specified here rather than as independent kwargs.

align : {'center', 'edge'}, optional

If 'edge', aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If 'center', interpret the `left` argument as the coordinates of the centers of the bars. To align on the align bars on the right edge pass a negative `width`.

orientation : {'vertical', 'horizontal'}, optional

The orientation of the bars.

log : boolean, optional

If true, sets the axis to be log scale. default: False

Returns bars : matplotlib.container.BarContainer

Container with all of the bars + errorbars

See also:

barh Plot a horizontal bar plot.

Notes

The optional arguments `color`, `edgecolor`, `linewidth`, `xerr`, and `yerr` can be either scalars or sequences of length equal to the number of bars. This enables you to use bar as the basis for stacked bar charts, or candlestick plots. Detail: `xerr` and `yerr` are passed directly to *errorbar()*, so they can also have shape 2xN for independent specification of lower and upper errors.

Other optional kwargs:

***papertype*:** One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

***format*:** One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

***transparent*:** If *True*, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

***frameon*:** If *True*, the figure patch will be colored, if *False*, the figure background will be transparent. If not provided, the rcParam 'savefig.frameon' will be used.

***bbox_inches*:** Bbox in inches. Only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

***pad_inches*:** Amount of padding around the figure when bbox_inches is 'tight'.

***bbox_extra_artists*:** A list of extra artists that will be considered when the tight bbox is calculated.

`matplotlib.pyplot.sca`(*ax*)

Set the current Axes instance to *ax*.

The current Figure is updated to the parent of *ax*.

`matplotlib.pyplot.scatter`(*x*, *y*, *s=None*, *c=None*, *marker=None*, *cmap=None*, *norm=None*, *vmin=None*, *vmax=None*, *alpha=None*, *linewidths=None*, *verts=None*, *edgecolors=None*, *hold=None*, *data=None*, ***kwargs*)

Make a scatter plot of `x` vs `y`

Marker size is scaled by `s` and marker color is mapped to `c`

Parameters x, y : array_like, shape (n,)

Input data

s : scalar or array_like, shape (n,), optional

size in points^2. Default is `rcParams['lines.markersize'] ** 2`.

c : color, sequence, or sequence of color, optional, default: 'b'

`c` can be a single color format string, or a sequence of color specifications of length `N`, or a sequence of `N` numbers to be mapped to colors using the `cmap` and `norm` specified via kwargs (see below). Note that `c` should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. `c` can be a 2-D array in which the rows are RGB or RGBA, however, including the case of a single row to specify the same color for all points.

marker : `MarkerStyle`, optional, default: 'o'

See `markers` for more information on the different styles of markers scatter supports. `marker` can be either an instance of the class or the text shorthand for a particular marker.

cmap : `Colormap`, optional, default: None

A `Colormap` instance or registered name. `cmap` is only used if `c` is an array of floats. If None, defaults to rc `image.cmap`.

norm : `Normalize`, optional, default: None

A `Normalize` instance is used to scale luminance data to 0, 1. `norm` is only used if `c` is an array of floats. If `None`, use the default `normalize()`.

vmin, vmax : scalar, optional, default: None

`vmin` and `vmax` are used in conjunction with `norm` to normalize luminance data. If either are `None`, the min and max of the color array is used. Note if you pass a `norm` instance, your settings for `vmin` and `vmax` will be ignored.

alpha : scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None

If None, defaults to (lines.linewidth,).

verts : sequence of (x, y), optional

If `marker` is None, these vertices will be used to construct the marker. The center of the marker is located at (0,0) in normalized units. The overall marker is rescaled by `s`.

edgecolors : color or sequence of color, optional, default: None

If None, defaults to 'face'

If 'face', the edge color will always be the same as the face color.

If it is 'none', the patch boundary will not be drawn.

For non-filled markers, the `edgecolors` kwarg is ignored and forced to 'face' internally.

Returns paths : `PathCollection`

Other Parameters kwargs : `Collection` properties

See also:

`plot` to plot scatter plots when markers are identical in size and color

Notes

- The `plot` function will be faster for scatterplots where markers don't vary in size or color.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

6.22.6 `sklearn.linear_model.Lasso`

class `sklearn.linear_model.`**`Lasso`**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *precompute=False*, *copy_X=True*, *max_iter=1000*, *tol=0.0001*, *warm_start=False*, *positive=False*, *random_state=None*, *selection='cyclic'*)

Linear Model trained with L1 prior as regularizer (aka the Lasso)

The optimization objective for Lasso is:

```
(1 / (2 * n_samples)) * ||y - Xw||^2_2 + alpha * ||w||_1
```

Technically the Lasso model is optimizing the same objective function as the Elastic Net with `l1_ratio=1.0` (no L2 penalty).

Read more in the *User Guide*.

Parameters

alpha [float, optional] Constant that multiplies the L1 term. Defaults to 1.0. `alpha = 0` is equivalent to an ordinary least square, solved by the *`LinearRegression`* object. For numerical reasons, using `alpha = 0` with the `Lasso` object is not advised. Given this, you should use the *`LinearRegression`* object.

fit_intercept [boolean, optional, default True] Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

precompute [True | False | array-like, default=False] Whether to use a precomputed Gram matrix to speed up calculations. If set to `'auto'` let us decide. The Gram matrix can also be passed as argument. For sparse input this option is always `True` to preserve sparsity.

copy_X [boolean, optional, default True] If `True`, X will be copied; else, it may be overwritten.

max_iter [int, optional] The maximum number of iterations

tol [float, optional] The tolerance for the optimization: if the updates are smaller than `tol`, the optimization code checks the dual gap for optimality and continues until it is smaller than `tol`.

warm_start [bool, optional] When set to True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. See *the Glossary*.

positive [bool, optional] When set to `True`, forces the coefficients to be positive.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator that selects a random feature to update. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `selection` == 'random'.

selection [str, default 'cyclic'] If set to 'random', a random coefficient is updated every iteration rather than looping over features sequentially by default. This (setting to 'random') often leads to significantly faster convergence especially when tol is higher than 1e-4.

Attributes

coef_ [array, shape (n_features,) | (n_targets, n_features)] parameter vector (w in the cost function formula)

sparse_coef_ [scipy.sparse matrix, shape (n_features, 1) | (n_targets, n_features)] sparse representation of the fitted `coef_`

intercept_ [float | array, shape (n_targets,)] independent term in decision function.

n_iter_ [int | array-like, shape (n_targets,)] number of iterations run by the coordinate descent solver to reach the specified tolerance.

See also:

lars_path

lasso_path

LassoLars

LassoCV

LassoLarsCV

sklearn.decomposition.sparse_encode

Notes

The algorithm used to fit the model is coordinate descent.

To avoid unnecessary memory duplication the X argument of the fit method should be directly passed as a Fortran-contiguous numpy array.

Examples

```
>>> from sklearn import linear_model
>>> clf = linear_model.Lasso(alpha=0.1)
>>> clf.fit([[0,0], [1, 1], [2, 2]], [0, 1, 2])
...
Lasso(alpha=0.1, copy_X=True, fit_intercept=True, max_iter=1000,
   normalize=False, positive=False, precompute=False, random_state=None,
   selection='cyclic', tol=0.0001, warm_start=False)
>>> print(clf.coef_)
[0.85 0.  ]
>>> print(clf.intercept_)
0.15...
```

Returns

C [array, shape (n_samples,)] Returns predicted values.

score (*self*, *X*, *y*, *sample_weight=None*)

Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples. For some estimators this may be a precomputed kernel matrix instead, shape = (n_samples, n_samples_fitted], where n_samples_fitted is the number of samples used in the fitting for the estimator.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

Notes

The R2 score used when calling `score` on a regressor will use `multioutput='uniform_average'` from version 0.23 to keep consistent with `metrics.r2_score`. This will influence the `score` method of all the multioutput regressors (except for `multioutput.MultiOutputRegressor`). To specify the default value manually and avoid the warning, please either call `metrics.r2_score` directly or make a custom scorer with `metrics.make_scorer` (the built-in scorer `'r2'` uses `multioutput='uniform_average'`).

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

6.22.8 sklearn.linear_model.LinearRegression

class `sklearn.linear_model.`**LinearRegression** (*fit_intercept=True*, *normalize=False*, *copy_X=True*, *n_jobs=None*)

Ordinary least squares Linear Regression.

Parameters

fit_intercept [boolean, optional, default True] whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

n_jobs [int or None, optional (default=None)] The number of jobs to use for the computation. This will only provide speedup for n_targets > 1 and sufficient large problems. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See *Glossary* for more details.

Attributes

coef_ [array, shape (n_features,) or (n_targets, n_features)] Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.

intercept_ [array] Independent term in the linear model.

Notes

From the implementation point of view, this is just plain Ordinary Least Squares (scipy.linalg.lstsq) wrapped as a predictor object.

Examples

```
>>> import numpy as np
>>> from sklearn.linear_model import LinearRegression
>>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
>>> # y = 1 * x_0 + 2 * x_1 + 3
>>> y = np.dot(X, np.array([1, 2])) + 3
>>> reg = LinearRegression().fit(X, y)
>>> reg.score(X, y)
1.0
>>> reg.coef_
array([1., 2.])
>>> reg.intercept_
3.0000...
>>> reg.predict(np.array([[3, 5]]))
array([16.])
```

Methods

`fit`(self, X, y[, sample_weight])	Fit linear model.
`get_params`(self[, deep])	Get parameters for this estimator.
`predict`(self, X)	Predict using the linear model
`score`(self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
`set_params`(self, **params)	Set the parameters of this estimator.

y [array, shape = [n_samples] or [n_samples, n_targets]] Target values.

Returns

z [float] Score of the prediction.

set_params (*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

Examples using `sklearn.linear_model.RANSACRegressor`

- *Robust linear model estimation using RANSAC*
- *Theil-Sen Regression*
- *Robust linear estimator fitting*

6.22.17 `sklearn.linear_model.Ridge`

class `sklearn.linear_model.`**`Ridge`**(*alpha=1.0*, *fit_intercept=True*, *normalize=False*, *copy_X=True*, *max_iter=None*, *tol=0.001*, *solver='auto'*, *random_state=None*)

Linear least squares with l2 regularization.

Minimizes the objective function:

```
||y - Xw||^2_2 + alpha * ||w||^2_2
```

This model solves a regression model where the loss function is the linear least squares function and regularization is given by the l2-norm. Also known as Ridge Regression or Tikhonov regularization. This estimator has built-in support for multi-variate regression (i.e., when y is a 2d-array of shape [n_samples, n_targets]).

Read more in the *User Guide*.

Parameters

alpha [{float, array-like}, shape (n_targets)] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Alpha corresponds to `C^-1` in other linear models such as LogisticRegression or LinearSVC. If an array is passed, penalties are assumed to be specific to the targets. Hence they must correspond in number.

fit_intercept [boolean] Whether to calculate the intercept for this model. If set to false, no intercept will be used in calculations (e.g. data is expected to be already centered).

normalize [boolean, optional, default False] This parameter is ignored when `fit_intercept` is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use *`sklearn.preprocessing.StandardScaler`* before calling `fit` on an estimator with `normalize=False`.

copy_X [boolean, optional, default True] If True, X will be copied; else, it may be overwritten.

max_iter [int, optional] Maximum number of iterations for conjugate gradient solver. For 'sparse_cg' and 'lsqr' solvers, the default value is determined by scipy.sparse.linalg. For 'sag' solver, the default value is 1000.

tol [float] Precision of the solution.

solver [{'auto', 'svd', 'cholesky', 'lsqr', 'sparse_cg', 'sag', 'saga'}] Solver to use in the computational routines:

- 'auto' chooses the solver automatically based on the type of data.
- 'svd' uses a Singular Value Decomposition of X to compute the Ridge coefficients. More stable for singular matrices than 'cholesky'.
- 'cholesky' uses the standard scipy.linalg.solve function to obtain a closed-form solution.
- 'sparse_cg' uses the conjugate gradient solver as found in scipy.sparse.linalg.cg. As an iterative algorithm, this solver is more appropriate than 'cholesky' for large-scale data (possibility to set `tol` and *max_iter*).
- 'lsqr' uses the dedicated regularized least-squares routine scipy.sparse.linalg.lsqr. It is the fastest and uses an iterative procedure.
- 'sag' uses a Stochastic Average Gradient descent, and 'saga' uses its improved, unbiased version named SAGA. Both methods also use an iterative procedure, and are often faster than other solvers when both n_samples and n_features are large. Note that 'sag' and 'saga' fast convergence is only guaranteed on features with approximately the same scale. You can preprocess the data with a scaler from sklearn.preprocessing.

All last five solvers support both dense and sparse data. However, only 'sag' and 'sparse_cg' supports sparse input when `fit_intercept` is True.

New in version 0.17: Stochastic Average Gradient descent solver.

New in version 0.19: SAGA solver.

random_state [int, RandomState instance or None, optional, default None] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`. Used when `solver` == 'sag'.

New in version 0.17: *random_state* to support Stochastic Average Gradient.

Attributes

coef_ [array, shape (n_features,) or (n_targets, n_features)] Weight vector(s).

intercept_ [float | array, shape = (n_targets,)] Independent term in decision function. Set to 0.0 if `fit_intercept = False`.

n_iter_ [array or None, shape (n_targets,)] Actual number of iterations for each target. Available only for sag and lsqr solvers. Other solvers will return None.

New in version 0.17.

See also:

`RidgeClassifier` Ridge classifier

`RidgeCV` Ridge regression with built-in cross validation

`sklearn.kernel_ridge.KernelRidge` Kernel ridge regression combines ridge regression with the kernel trick

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] If multioutput is 'raw_values', then mean absolute error is returned for each output separately. If multioutput is 'uniform_average' or an ndarray of weights, then the weighted average of all output errors is returned.

MAE output is non-negative floating point. The best value is 0.0.

Examples

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_absolute_error(y_true, y_pred)
0.5
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> mean_absolute_error(y_true, y_pred)
0.75
>>> mean_absolute_error(y_true, y_pred, multioutput='raw_values')
array([0.5, 1. ])
>>> mean_absolute_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.85...
```

sklearn.metrics.mean_squared_error

`sklearn.metrics.`**`mean_squared_error`**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean squared error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average']] or array-like of shape (n_outputs) Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

'raw_values' : Returns a full set of errors in case of multioutput input.

'uniform_average' : Errors of all outputs are averaged with uniform weight.

Returns

loss [float or ndarray of floats] A non-negative floating point value (the best value is 0.0), or an array of floating point values, one for each individual target.

Examples

```
>>> from sklearn.metrics import mean_squared_error
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> mean_squared_error(y_true, y_pred)
0.375
>>> y_true = [[0.5, 1],[-1, 1],[7, -6]]
>>> y_pred = [[0, 2],[-1, 2],[8, -5]]
>>> mean_squared_error(y_true, y_pred)
0.708...
>>> mean_squared_error(y_true, y_pred, multioutput='raw_values')
...
array([0.41666667, 1.        ])
>>> mean_squared_error(y_true, y_pred, multioutput=[0.3, 0.7])
...
0.825...
```

Examples using `sklearn.metrics.mean_squared_error`

- *Model Complexity Influence*
- *Gradient Boosting regression*
- *Plot Ridge coefficients as a function of the L2 regularization*
- *Linear Regression Example*
- *Robust linear estimator fitting*

`sklearn.metrics`.mean_squared_log_error

`sklearn.metrics.`**`mean_squared_log_error`**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

Mean squared logarithmic error regression loss

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average'] or array-like of shape = (n_outputs)] Defines aggregating of multiple output values. Array-like value defines weights used to average errors.

sklearn.metrics.r2_score

sklearn.metrics.**r2_score**(*y_true*, *y_pred*, *sample_weight=None*, *multioutput='uniform_average'*)

R^2 (coefficient of determination) regression score function.

Best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Read more in the *User Guide*.

Parameters

y_true [array-like of shape = (n_samples) or (n_samples, n_outputs)] Ground truth (correct) target values.

y_pred [array-like of shape = (n_samples) or (n_samples, n_outputs)] Estimated target values.

sample_weight [array-like of shape = (n_samples), optional] Sample weights.

multioutput [string in ['raw_values', 'uniform_average', 'variance_weighted'] or None or array-like of shape (n_outputs)] Defines aggregating of multiple output scores. Array-like value defines weights used to average scores. Default is "uniform_average".

'raw_values' : Returns a full set of scores in case of multioutput input.

'uniform_average' : Scores of all outputs are averaged with uniform weight.

'variance_weighted' : Scores of all outputs are averaged, weighted by the variances of each individual output.

Changed in version 0.19: Default value of multioutput is 'uniform_average'.

Returns

z [float or ndarray of floats] The R^2 score or ndarray of scores if 'multioutput' is 'raw_values'.

Notes

This is not a symmetric function.

Unlike most other scores, R^2 score may be negative (it need not actually be the square of a quantity R).

This metric is not well-defined for single samples and will return a NaN value if n_samples is less than two.

References

[1]

Examples

```
>>> from sklearn.metrics import r2_score
>>> y_true = [3, -0.5, 2, 7]
>>> y_pred = [2.5, 0.0, 2, 8]
>>> r2_score(y_true, y_pred)
0.948...
>>> y_true = [[0.5, 1], [-1, 1], [7, -6]]
>>> y_pred = [[0, 2], [-1, 2], [8, -5]]
>>> r2_score(y_true, y_pred,
...          multioutput='variance_weighted')
```

cv [int, cross-validation generator or an iterable, optional] Determines the cross-validation splitting strategy. Possible inputs for cv are:

- None, to use the default 3-fold cross-validation,
- integer, to specify the number of folds.
- *CV splitter*,
- An iterable yielding (train, test) splits as arrays of indices.

For integer/None inputs, if classifier is True and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used.

Refer *User Guide* for the various cross-validation strategies that can be used here.

Changed in version 0.20: `cv` default value will change from 3-fold to 5-fold in v0.22.

y [array-like, optional] The target variable for supervised learning problems.

classifier [boolean, optional, default False] Whether the task is a classification task, in which case stratified KFold will be used.

Returns

checked_cv [a cross-validator instance.] The return value is a cross-validator which generates the train/test splits via the `split` method.

`sklearn.model_selection`.train_test_split

`sklearn.model_selection.`**`train_test_split`**(**arrays*, ***options*)

Split arrays or matrices into random train and test subsets

Quick utility that wraps input validation and `next(ShuffleSplit().split(X, y))` and application to input data into a single call for splitting (and optionally subsampling) data in a oneliner.

Read more in the *User Guide*.

Parameters

***arrays** [sequence of indexables with same length / shape[0]] Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size [float, int or None, optional (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If `train_size` is also None, it will be set to 0.25.

train_size [float, int, or None, (default=None)] If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by `np.random`.

shuffle [boolean, optional (default=True)] Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify [array-like or None (default=None)] If not None, data is split in a stratified fashion, using this as the class labels.

X_new [numpy array of shape [n_samples, n_features_new]] Transformed array.

get_params(*self*, *deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

inverse_transform(*self*, *X*)

Scale back the data to the original representation

Parameters

X [{array-like, sparse matrix}] The data that should be transformed back.

partial_fit(*self*, *X*, *y=None*)

Online computation of max absolute value of X for later scaling. All of X is processed as a single batch. This is intended for cases when `fit` is not feasible due to very large number of *n_samples* or because X is read from a continuous stream.

Parameters

X [{array-like, sparse matrix}, shape [n_samples, n_features]] The data used to compute the mean and standard deviation used for later scaling along the features axis.

y Ignored

set_params(*self*, ***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

transform(*self*, *X*)

Scale the data

Parameters

X [{array-like, sparse matrix}] The data that should be scaled.

Examples using `sklearn.preprocessing.MaxAbsScaler`

- *Compare the effect of different scalers on data with outliers*

6.34.9 `sklearn.preprocessing`.MinMaxScaler

class `sklearn.preprocessing.`**MinMaxScaler**(*feature_range=(0, 1)*, *copy=True*)

Transforms features by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one.

The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

The transformation is calculated as:

```
X_scaled = scale * X + min - X.min(axis=0) * scale
where scale = (max - min) / (X.max(axis=0) - X.min(axis=0))
```

This transformation is often used as an alternative to zero mean, unit variance scaling.

Read more in the *User Guide*.

Parameters

feature_range [tuple (min, max), default=(0, 1)] Desired range of transformed data.

copy [boolean, optional, default True] Set to False to perform inplace row normalization and avoid a copy (if the input is already a numpy array).

Attributes

min_ [ndarray, shape (n_features,)] Per feature adjustment for minimum. Equivalent to `min - X.min(axis=0) * self.scale_`

scale_ [ndarray, shape (n_features,)] Per feature relative scaling of the data. Equivalent to `(max - min) / (X.max(axis=0) - X.min(axis=0))`

New in version 0.17: *scale_* attribute.

data_min_ [ndarray, shape (n_features,)] Per feature minimum seen in the data

New in version 0.17: *data_min_*

data_max_ [ndarray, shape (n_features,)] Per feature maximum seen in the data

New in version 0.17: *data_max_*

data_range_ [ndarray, shape (n_features,)] Per feature range `(data_max_ - data_min_)` seen in the data

New in version 0.17: *data_range_*

See also:

`minmax_scale` Equivalent function without the estimator API.

Notes

NaNs are treated as missing values: disregarded in fit, and maintained in transform.

For a comparison of the different scalers, transformers, and normalizers, see *examples/preprocessing/plot_all_scaling.py*.

Examples