LinearRegression源码

```
class LinearRegression(LinearModel, RegressorMixin):
    """
    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
        :class:`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, optional, default 1
        The number of jobs to use for the
computation.
        If -1 all CPUs are used. This will only
```

```
provide speedup for
        n_targets > 1 and sufficient large problems.

    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.

    """

    def __init__(self, fit_intercept=True,
normalize=False, copy_X=True,
                 n_jobs=1):
        self.fit_intercept = fit_intercept
        self.normalize = normalize
        self.copy_X = copy_X
        self.n_jobs = n_jobs

    def fit(self, X, y, sample_weight=None):
        """
        Fit linear model.
```

```
        Parameters
        ----------
        X : numpy array or sparse matrix of shape
[n_samples,n_features]
            Training data

        y : numpy array of shape [n_samples,
n_targets]
            Target values. Will be cast to X's dtype
if necessary

        sample_weight : numpy array of shape
[n_samples]
            Individual weights for each sample

            .. versionadded:: 0.17
               parameter *sample_weight* support to
LinearRegression.

        Returns
        -------
        self : returns an instance of self.
        """

        n_jobs_ = self.n_jobs
        X, y = check_X_y(X, y, accept_sparse=['csr',
'csc', 'coo'],
                         y_numeric=True,
multi_output=True)

        if sample_weight is not None and
np.atleast_1d(sample_weight).ndim > 1:
            raise ValueError("Sample weights must be
1D array or scalar")

        X, y, X_offset, y_offset, X_scale =
self._preprocess_data(
            X, y, fit_intercept=self.fit_intercept,
```

```
normalize=self.normalize,
            copy=self.copy_X,
sample_weight=sample_weight)

        if sample_weight is not None:
            # Sample weight can be implemented via a
simple rescaling.
            X, y = _rescale_data(X, y, sample_weight)

        if sp.issparse(X):
            if y.ndim < 2:
                out = sparse_lsqr(X, y)
                self.coef_ = out[0]
                self._residues = out[3]
            else:
                # sparse_lstsq cannot handle y with
shape (M, K)
                outs = Parallel(n_jobs=n_jobs_)(
                    delayed(sparse_lsqr)(X, y[:,
j].ravel())
                    for j in range(y.shape[1]))
                self.coef_ = np.vstack(out[0] for out
in outs)
                self._residues = np.vstack(out[3] for
out in outs)
        else:
            self.coef_, self._residues, self.rank_,
self.singular_ = \
                linalg.lstsq(X, y)
            self.coef_ = self.coef_.T

        if y.ndim == 1:
            self.coef_ = np.ravel(self.coef_)
        self._set_intercept(X_offset, y_offset,
X_scale)
        return self


    class LinearModel(six.with_metaclass(ABCMeta,
```

```
BaseEstimator)):
        """Base class for Linear Models"""
        @abstractmethod
        def fit(self, X, y):
            """Fit model."""

        def _decision_function(self, X):
            check_is_fitted(self, "coef_")

            X = check_array(X, accept_sparse=['csr',
'csc', 'coo'])
            return safe_sparse_dot(X, self.coef_.T,
                                   dense_output=True)
+ self.intercept_

        def predict(self, X):
            """Predict using the linear model

            Parameters
            ----------
            X : {array-like, sparse matrix}, shape =
(n_samples, n_features)
                Samples.

            Returns
            -------
            C : array, shape = (n_samples,)
                Returns predicted values.
            """
            return self._decision_function(X)

        _preprocess_data =
staticmethod(_preprocess_data)

        def _set_intercept(self, X_offset, y_offset,
X_scale):
            """Set the intercept_
            """
            if self.fit_intercept:
```

```
                self.coef_ = self.coef_ / X_scale
                self.intercept_ = y_offset -
np.dot(X_offset, self.coef_.T)
            else:
                self.intercept_ = 0.
```