"""

Logistic Regression

"""

# Author: Gael Varoquaux <gael.varoquaux@normalesup.org>

# Fabian Pedregosa <f@bianp.net>

# Alexandre Gramfort <alexandre.gramfort@telecom-paristech.fr>

# Manoj Kumar <manojkumarsivaraj334@gmail.com>

# Lars Buitinck

# Simon Wu <s8wu@uwaterloo.ca>

# Arthur Mensch <arthur.mensch@m4x.org

import numbers

import warnings

import numpy as np

from scipy import optimize, sparse

from scipy.special import expit

from .base import LinearClassifierMixin, SparseCoefMixin, BaseEstimator

from .sag import sag\_solver

from ..preprocessing import LabelEncoder, LabelBinarizer

from ..svm.base import \_fit\_liblinear

from ..utils import check\_array, check\_consistent\_length, compute\_class\_weight

from ..utils import check\_random\_state

from ..utils.extmath import (log\_logistic, safe\_sparse\_dot, softmax,

squared\_norm)

from ..utils.extmath import row\_norms

from ..utils.fixes import logsumexp

from ..utils.optimize import newton\_cg

from ..utils.validation import check\_X\_y

from ..exceptions import (NotFittedError, ConvergenceWarning,

ChangedBehaviorWarning)

from ..utils.multiclass import check\_classification\_targets

from ..utils.\_joblib import Parallel, delayed, effective\_n\_jobs

from ..utils.fixes import \_joblib\_parallel\_args

from ..model\_selection import check\_cv

from ..externals import six

from ..metrics import get\_scorer

# .. some helper functions for logistic\_regression\_path ..

def \_intercept\_dot(w, X, y):

"""Computes y \* np.dot(X, w).

It takes into consideration if the intercept should be fit or not.

Parameters

----------

w : ndarray, shape (n\_features,) or (n\_features + 1,)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

y : ndarray, shape (n\_samples,)

Array of labels.

Returns

-------

w : ndarray, shape (n\_features,)

Coefficient vector without the intercept weight (w[-1]) if the

intercept should be fit. Unchanged otherwise.

c : float

The intercept.

yz : float

y \* np.dot(X, w).

"""

c = 0.

if w.size == X.shape[1] + 1:

c = w[-1]

w = w[:-1]

z = safe\_sparse\_dot(X, w) + c

yz = y \* z

return w, c, yz

def \_logistic\_loss\_and\_grad(w, X, y, alpha, sample\_weight=None):

"""Computes the logistic loss and gradient.

Parameters

----------

w : ndarray, shape (n\_features,) or (n\_features + 1,)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

y : ndarray, shape (n\_samples,)

Array of labels.

alpha : float

Regularization parameter. alpha is equal to 1 / C.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

out : float

Logistic loss.

grad : ndarray, shape (n\_features,) or (n\_features + 1,)

Logistic gradient.

"""

n\_samples, n\_features = X.shape

grad = np.empty\_like(w)

w, c, yz = \_intercept\_dot(w, X, y)

if sample\_weight is None:

sample\_weight = np.ones(n\_samples)

# Logistic loss is the negative of the log of the logistic function.

out = -np.sum(sample\_weight \* log\_logistic(yz)) + .5 \* alpha \* np.dot(w, w)

z = expit(yz)

z0 = sample\_weight \* (z - 1) \* y

grad[:n\_features] = safe\_sparse\_dot(X.T, z0) + alpha \* w

# Case where we fit the intercept.

if grad.shape[0] > n\_features:

grad[-1] = z0.sum()

return out, grad

def \_logistic\_loss(w, X, y, alpha, sample\_weight=None):

"""Computes the logistic loss.

Parameters

----------

w : ndarray, shape (n\_features,) or (n\_features + 1,)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

y : ndarray, shape (n\_samples,)

Array of labels.

alpha : float

Regularization parameter. alpha is equal to 1 / C.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

out : float

Logistic loss.

"""

w, c, yz = \_intercept\_dot(w, X, y)

if sample\_weight is None:

sample\_weight = np.ones(y.shape[0])

# Logistic loss is the negative of the log of the logistic function.

out = -np.sum(sample\_weight \* log\_logistic(yz)) + .5 \* alpha \* np.dot(w, w)

return out

def \_logistic\_grad\_hess(w, X, y, alpha, sample\_weight=None):

"""Computes the gradient and the Hessian, in the case of a logistic loss.

Parameters

----------

w : ndarray, shape (n\_features,) or (n\_features + 1,)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

y : ndarray, shape (n\_samples,)

Array of labels.

alpha : float

Regularization parameter. alpha is equal to 1 / C.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

grad : ndarray, shape (n\_features,) or (n\_features + 1,)

Logistic gradient.

Hs : callable

Function that takes the gradient as a parameter and returns the

matrix product of the Hessian and gradient.

"""

n\_samples, n\_features = X.shape

grad = np.empty\_like(w)

fit\_intercept = grad.shape[0] > n\_features

w, c, yz = \_intercept\_dot(w, X, y)

if sample\_weight is None:

sample\_weight = np.ones(y.shape[0])

z = expit(yz)

z0 = sample\_weight \* (z - 1) \* y

grad[:n\_features] = safe\_sparse\_dot(X.T, z0) + alpha \* w

# Case where we fit the intercept.

if fit\_intercept:

grad[-1] = z0.sum()

# The mat-vec product of the Hessian

d = sample\_weight \* z \* (1 - z)

if sparse.issparse(X):

dX = safe\_sparse\_dot(sparse.dia\_matrix((d, 0),

shape=(n\_samples, n\_samples)), X)

else:

# Precompute as much as possible

dX = d[:, np.newaxis] \* X

if fit\_intercept:

# Calculate the double derivative with respect to intercept

# In the case of sparse matrices this returns a matrix object.

dd\_intercept = np.squeeze(np.array(dX.sum(axis=0)))

def Hs(s):

ret = np.empty\_like(s)

ret[:n\_features] = X.T.dot(dX.dot(s[:n\_features]))

ret[:n\_features] += alpha \* s[:n\_features]

# For the fit intercept case.

if fit\_intercept:

ret[:n\_features] += s[-1] \* dd\_intercept

ret[-1] = dd\_intercept.dot(s[:n\_features])

ret[-1] += d.sum() \* s[-1]

return ret

return grad, Hs

def \_multinomial\_loss(w, X, Y, alpha, sample\_weight):

"""Computes multinomial loss and class probabilities.

Parameters

----------

w : ndarray, shape (n\_classes \* n\_features,) or

(n\_classes \* (n\_features + 1),)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

Y : ndarray, shape (n\_samples, n\_classes)

Transformed labels according to the output of LabelBinarizer.

alpha : float

Regularization parameter. alpha is equal to 1 / C.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

loss : float

Multinomial loss.

p : ndarray, shape (n\_samples, n\_classes)

Estimated class probabilities.

w : ndarray, shape (n\_classes, n\_features)

Reshaped param vector excluding intercept terms.

Reference

---------

Bishop, C. M. (2006). Pattern recognition and machine learning.

Springer. (Chapter 4.3.4)

"""

n\_classes = Y.shape[1]

n\_features = X.shape[1]

fit\_intercept = w.size == (n\_classes \* (n\_features + 1))

w = w.reshape(n\_classes, -1)

sample\_weight = sample\_weight[:, np.newaxis]

if fit\_intercept:

intercept = w[:, -1]

w = w[:, :-1]

else:

intercept = 0

p = safe\_sparse\_dot(X, w.T)

p += intercept

p -= logsumexp(p, axis=1)[:, np.newaxis]

loss = -(sample\_weight \* Y \* p).sum()

loss += 0.5 \* alpha \* squared\_norm(w)

p = np.exp(p, p)

return loss, p, w

def \_multinomial\_loss\_grad(w, X, Y, alpha, sample\_weight):

"""Computes the multinomial loss, gradient and class probabilities.

Parameters

----------

w : ndarray, shape (n\_classes \* n\_features,) or

(n\_classes \* (n\_features + 1),)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

Y : ndarray, shape (n\_samples, n\_classes)

Transformed labels according to the output of LabelBinarizer.

alpha : float

Regularization parameter. alpha is equal to 1 / C.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

Returns

-------

loss : float

Multinomial loss.

grad : ndarray, shape (n\_classes \* n\_features,) or

(n\_classes \* (n\_features + 1),)

Ravelled gradient of the multinomial loss.

p : ndarray, shape (n\_samples, n\_classes)

Estimated class probabilities

Reference

---------

Bishop, C. M. (2006). Pattern recognition and machine learning.

Springer. (Chapter 4.3.4)

"""

n\_classes = Y.shape[1]

n\_features = X.shape[1]

fit\_intercept = (w.size == n\_classes \* (n\_features + 1))

grad = np.zeros((n\_classes, n\_features + bool(fit\_intercept)),

dtype=X.dtype)

loss, p, w = \_multinomial\_loss(w, X, Y, alpha, sample\_weight)

sample\_weight = sample\_weight[:, np.newaxis]

diff = sample\_weight \* (p - Y)

grad[:, :n\_features] = safe\_sparse\_dot(diff.T, X)

grad[:, :n\_features] += alpha \* w

if fit\_intercept:

grad[:, -1] = diff.sum(axis=0)

return loss, grad.ravel(), p

def \_multinomial\_grad\_hess(w, X, Y, alpha, sample\_weight):

"""

Computes the gradient and the Hessian, in the case of a multinomial loss.

Parameters

----------

w : ndarray, shape (n\_classes \* n\_features,) or

(n\_classes \* (n\_features + 1),)

Coefficient vector.

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

Y : ndarray, shape (n\_samples, n\_classes)

Transformed labels according to the output of LabelBinarizer.

alpha : float

Regularization parameter. alpha is equal to 1 / C.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

Returns

-------

grad : array, shape (n\_classes \* n\_features,) or

(n\_classes \* (n\_features + 1),)

Ravelled gradient of the multinomial loss.

hessp : callable

Function that takes in a vector input of shape (n\_classes \* n\_features)

or (n\_classes \* (n\_features + 1)) and returns matrix-vector product

with hessian.

References

----------

Barak A. Pearlmutter (1993). Fast Exact Multiplication by the Hessian.

http://www.bcl.hamilton.ie/~barak/papers/nc-hessian.pdf

"""

n\_features = X.shape[1]

n\_classes = Y.shape[1]

fit\_intercept = w.size == (n\_classes \* (n\_features + 1))

# `loss` is unused. Refactoring to avoid computing it does not

# significantly speed up the computation and decreases readability

loss, grad, p = \_multinomial\_loss\_grad(w, X, Y, alpha, sample\_weight)

sample\_weight = sample\_weight[:, np.newaxis]

# Hessian-vector product derived by applying the R-operator on the gradient

# of the multinomial loss function.

def hessp(v):

v = v.reshape(n\_classes, -1)

if fit\_intercept:

inter\_terms = v[:, -1]

v = v[:, :-1]

else:

inter\_terms = 0

# r\_yhat holds the result of applying the R-operator on the multinomial

# estimator.

r\_yhat = safe\_sparse\_dot(X, v.T)

r\_yhat += inter\_terms

r\_yhat += (-p \* r\_yhat).sum(axis=1)[:, np.newaxis]

r\_yhat \*= p

r\_yhat \*= sample\_weight

hessProd = np.zeros((n\_classes, n\_features + bool(fit\_intercept)))

hessProd[:, :n\_features] = safe\_sparse\_dot(r\_yhat.T, X)

hessProd[:, :n\_features] += v \* alpha

if fit\_intercept:

hessProd[:, -1] = r\_yhat.sum(axis=0)

return hessProd.ravel()

return grad, hessp

def \_check\_solver(solver, penalty, dual):

if solver == 'warn':

solver = 'liblinear'

warnings.warn("Default solver will be changed to 'lbfgs' in 0.22. "

"Specify a solver to silence this warning.",

FutureWarning)

all\_solvers = ['liblinear', 'newton-cg', 'lbfgs', 'sag', 'saga']

if solver not in all\_solvers:

raise ValueError("Logistic Regression supports only solvers in %s, got"

" %s." % (all\_solvers, solver))

all\_penalties = ['l1', 'l2']

if penalty not in all\_penalties:

raise ValueError("Logistic Regression supports only penalties in %s,"

" got %s." % (all\_penalties, penalty))

if solver not in ['liblinear', 'saga'] and penalty != 'l2':

raise ValueError("Solver %s supports only l2 penalties, "

"got %s penalty." % (solver, penalty))

if solver != 'liblinear' and dual:

raise ValueError("Solver %s supports only "

"dual=False, got dual=%s" % (solver, dual))

return solver

def \_check\_multi\_class(multi\_class, solver, n\_classes):

if multi\_class == 'warn':

multi\_class = 'ovr'

if n\_classes > 2:

warnings.warn("Default multi\_class will be changed to 'auto' in"

" 0.22. Specify the multi\_class option to silence "

"this warning.", FutureWarning)

if multi\_class == 'auto':

if solver == 'liblinear':

multi\_class = 'ovr'

elif n\_classes > 2:

multi\_class = 'multinomial'

else:

multi\_class = 'ovr'

if multi\_class not in ('multinomial', 'ovr'):

raise ValueError("multi\_class should be 'multinomial', 'ovr' or "

"'auto'. Got %s." % multi\_class)

if multi\_class == 'multinomial' and solver == 'liblinear':

raise ValueError("Solver %s does not support "

"a multinomial backend." % solver)

return multi\_class

def logistic\_regression\_path(X, y, pos\_class=None, Cs=10, fit\_intercept=True,

max\_iter=100, tol=1e-4, verbose=0,

solver='lbfgs', coef=None,

class\_weight=None, dual=False, penalty='l2',

intercept\_scaling=1., multi\_class='warn',

random\_state=None, check\_input=True,

max\_squared\_sum=None, sample\_weight=None):

"""Compute a Logistic Regression model for a list of regularization

parameters.

This is an implementation that uses the result of the previous model

to speed up computations along the set of solutions, making it faster

than sequentially calling LogisticRegression for the different parameters.

Note that there will be no speedup with liblinear solver, since it does

not handle warm-starting.

Read more in the :ref:`User Guide <logistic\_regression>`.

Parameters

----------

X : array-like or sparse matrix, shape (n\_samples, n\_features)

Input data.

y : array-like, shape (n\_samples,) or (n\_samples, n\_targets)

Input data, target values.

pos\_class : int, None

The class with respect to which we perform a one-vs-all fit.

If None, then it is assumed that the given problem is binary.

Cs : int | array-like, shape (n\_cs,)

List of values for the regularization parameter or integer specifying

the number of regularization parameters that should be used. In this

case, the parameters will be chosen in a logarithmic scale between

1e-4 and 1e4.

fit\_intercept : bool

Whether to fit an intercept for the model. In this case the shape of

the returned array is (n\_cs, n\_features + 1).

max\_iter : int

Maximum number of iterations for the solver.

tol : float

Stopping criterion. For the newton-cg and lbfgs solvers, the iteration

will stop when ``max{|g\_i | i = 1, ..., n} <= tol``

where ``g\_i`` is the i-th component of the gradient.

verbose : int

For the liblinear and lbfgs solvers set verbose to any positive

number for verbosity.

solver : {'lbfgs', 'newton-cg', 'liblinear', 'sag', 'saga'}

Numerical solver to use.

coef : array-like, shape (n\_features,), default None

Initialization value for coefficients of logistic regression.

Useless for liblinear solver.

class\_weight : dict or 'balanced', optional

Weights associated with classes in the form ``{class\_label: weight}``.

If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust

weights inversely proportional to class frequencies in the input data

as ``n\_samples / (n\_classes \* np.bincount(y))``.

Note that these weights will be multiplied with sample\_weight (passed

through the fit method) if sample\_weight is specified.

dual : bool

Dual or primal formulation. Dual formulation is only implemented for

l2 penalty with liblinear solver. Prefer dual=False when

n\_samples > n\_features.

penalty : str, 'l1' or 'l2'

Used to specify the norm used in the penalization. The 'newton-cg',

'sag' and 'lbfgs' solvers support only l2 penalties.

intercept\_scaling : float, default 1.

Useful only when the solver 'liblinear' is used

and self.fit\_intercept is set to True. In this case, x becomes

[x, self.intercept\_scaling],

i.e. a "synthetic" feature with constant value equal to

intercept\_scaling is appended to the instance vector.

The intercept becomes ``intercept\_scaling \* synthetic\_feature\_weight``.

Note! the synthetic feature weight is subject to l1/l2 regularization

as all other features.

To lessen the effect of regularization on synthetic feature weight

(and therefore on the intercept) intercept\_scaling has to be increased.

multi\_class : str, {'ovr', 'multinomial', 'auto'}, default: 'ovr'

If the option chosen is 'ovr', then a binary problem is fit for each

label. For 'multinomial' the loss minimised is the multinomial loss fit

across the entire probability distribution, \*even when the data is

binary\*. 'multinomial' is unavailable when solver='liblinear'.

'auto' selects 'ovr' if the data is binary, or if solver='liblinear',

and otherwise selects 'multinomial'.

.. versionadded:: 0.18

Stochastic Average Gradient descent solver for 'multinomial' case.

.. versionchanged:: 0.20

Default will change from 'ovr' to 'auto' in 0.22.

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' or

'liblinear'.

check\_input : bool, default True

If False, the input arrays X and y will not be checked.

max\_squared\_sum : float, default None

Maximum squared sum of X over samples. Used only in SAG solver.

If None, it will be computed, going through all the samples.

The value should be precomputed to speed up cross validation.

sample\_weight : array-like, shape(n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

coefs : ndarray, shape (n\_cs, n\_features) or (n\_cs, n\_features + 1)

List of coefficients for the Logistic Regression model. If

fit\_intercept is set to True then the second dimension will be

n\_features + 1, where the last item represents the intercept. For

``multiclass='multinomial'``, the shape is (n\_classes, n\_cs,

n\_features) or (n\_classes, n\_cs, n\_features + 1).

Cs : ndarray

Grid of Cs used for cross-validation.

n\_iter : array, shape (n\_cs,)

Actual number of iteration for each Cs.

Notes

-----

You might get slightly different results with the solver liblinear than

with the others since this uses LIBLINEAR which penalizes the intercept.

.. versionchanged:: 0.19

The "copy" parameter was removed.

"""

if isinstance(Cs, numbers.Integral):

Cs = np.logspace(-4, 4, Cs)

solver = \_check\_solver(solver, penalty, dual)

# Preprocessing.

if check\_input:

X = check\_array(X, accept\_sparse='csr', dtype=np.float64,

accept\_large\_sparse=solver != 'liblinear')

y = check\_array(y, ensure\_2d=False, dtype=None)

check\_consistent\_length(X, y)

\_, n\_features = X.shape

classes = np.unique(y)

random\_state = check\_random\_state(random\_state)

multi\_class = \_check\_multi\_class(multi\_class, solver, len(classes))

if pos\_class is None and multi\_class != 'multinomial':

if (classes.size > 2):

raise ValueError('To fit OvR, use the pos\_class argument')

# np.unique(y) gives labels in sorted order.

pos\_class = classes[1]

# If sample weights exist, convert them to array (support for lists)

# and check length

# Otherwise set them to 1 for all examples

if sample\_weight is not None:

sample\_weight = np.array(sample\_weight, dtype=X.dtype, order='C')

check\_consistent\_length(y, sample\_weight)

else:

sample\_weight = np.ones(X.shape[0], dtype=X.dtype)

# If class\_weights is a dict (provided by the user), the weights

# are assigned to the original labels. If it is "balanced", then

# the class\_weights are assigned after masking the labels with a OvR.

le = LabelEncoder()

if isinstance(class\_weight, dict) or multi\_class == 'multinomial':

class\_weight\_ = compute\_class\_weight(class\_weight, classes, y)

sample\_weight \*= class\_weight\_[le.fit\_transform(y)]

# For doing a ovr, we need to mask the labels first. for the

# multinomial case this is not necessary.

if multi\_class == 'ovr':

w0 = np.zeros(n\_features + int(fit\_intercept), dtype=X.dtype)

mask\_classes = np.array([-1, 1])

mask = (y == pos\_class)

y\_bin = np.ones(y.shape, dtype=X.dtype)

y\_bin[~mask] = -1.

# for compute\_class\_weight

if class\_weight == "balanced":

class\_weight\_ = compute\_class\_weight(class\_weight, mask\_classes,

y\_bin)

sample\_weight \*= class\_weight\_[le.fit\_transform(y\_bin)]

else:

if solver not in ['sag', 'saga']:

lbin = LabelBinarizer()

Y\_multi = lbin.fit\_transform(y)

if Y\_multi.shape[1] == 1:

Y\_multi = np.hstack([1 - Y\_multi, Y\_multi])

else:

# SAG multinomial solver needs LabelEncoder, not LabelBinarizer

le = LabelEncoder()

Y\_multi = le.fit\_transform(y).astype(X.dtype, copy=False)

w0 = np.zeros((classes.size, n\_features + int(fit\_intercept)),

order='F', dtype=X.dtype)

if coef is not None:

# it must work both giving the bias term and not

if multi\_class == 'ovr':

if coef.size not in (n\_features, w0.size):

raise ValueError(

'Initialization coef is of shape %d, expected shape '

'%d or %d' % (coef.size, n\_features, w0.size))

w0[:coef.size] = coef

else:

# For binary problems coef.shape[0] should be 1, otherwise it

# should be classes.size.

n\_classes = classes.size

if n\_classes == 2:

n\_classes = 1

if (coef.shape[0] != n\_classes or

coef.shape[1] not in (n\_features, n\_features + 1)):

raise ValueError(

'Initialization coef is of shape (%d, %d), expected '

'shape (%d, %d) or (%d, %d)' % (

coef.shape[0], coef.shape[1], classes.size,

n\_features, classes.size, n\_features + 1))

if n\_classes == 1:

w0[0, :coef.shape[1]] = -coef

w0[1, :coef.shape[1]] = coef

else:

w0[:, :coef.shape[1]] = coef

if multi\_class == 'multinomial':

# fmin\_l\_bfgs\_b and newton-cg accepts only ravelled parameters.

if solver in ['lbfgs', 'newton-cg']:

w0 = w0.ravel()

target = Y\_multi

if solver == 'lbfgs':

func = lambda x, \*args: \_multinomial\_loss\_grad(x, \*args)[0:2]

elif solver == 'newton-cg':

func = lambda x, \*args: \_multinomial\_loss(x, \*args)[0]

grad = lambda x, \*args: \_multinomial\_loss\_grad(x, \*args)[1]

hess = \_multinomial\_grad\_hess

warm\_start\_sag = {'coef': w0.T}

else:

target = y\_bin

if solver == 'lbfgs':

func = \_logistic\_loss\_and\_grad

elif solver == 'newton-cg':

func = \_logistic\_loss

grad = lambda x, \*args: \_logistic\_loss\_and\_grad(x, \*args)[1]

hess = \_logistic\_grad\_hess

warm\_start\_sag = {'coef': np.expand\_dims(w0, axis=1)}

coefs = list()

n\_iter = np.zeros(len(Cs), dtype=np.int32)

for i, C in enumerate(Cs):

if solver == 'lbfgs':

iprint = [-1, 50, 1, 100, 101][

np.searchsorted(np.array([0, 1, 2, 3]), verbose)]

w0, loss, info = optimize.fmin\_l\_bfgs\_b(

func, w0, fprime=None,

args=(X, target, 1. / C, sample\_weight),

iprint=iprint, pgtol=tol, maxiter=max\_iter)

if info["warnflag"] == 1:

warnings.warn("lbfgs failed to converge. Increase the number "

"of iterations.", ConvergenceWarning)

# In scipy <= 1.0.0, nit may exceed maxiter.

# See https://github.com/scipy/scipy/issues/7854.

n\_iter\_i = min(info['nit'], max\_iter)

elif solver == 'newton-cg':

args = (X, target, 1. / C, sample\_weight)

w0, n\_iter\_i = newton\_cg(hess, func, grad, w0, args=args,

maxiter=max\_iter, tol=tol)

elif solver == 'liblinear':

coef\_, intercept\_, n\_iter\_i, = \_fit\_liblinear(

X, target, C, fit\_intercept, intercept\_scaling, None,

penalty, dual, verbose, max\_iter, tol, random\_state,

sample\_weight=sample\_weight)

if fit\_intercept:

w0 = np.concatenate([coef\_.ravel(), intercept\_])

else:

w0 = coef\_.ravel()

elif solver in ['sag', 'saga']:

if multi\_class == 'multinomial':

target = target.astype(np.float64)

loss = 'multinomial'

else:

loss = 'log'

if penalty == 'l1':

alpha = 0.

beta = 1. / C

else:

alpha = 1. / C

beta = 0.

w0, n\_iter\_i, warm\_start\_sag = sag\_solver(

X, target, sample\_weight, loss, alpha,

beta, max\_iter, tol,

verbose, random\_state, False, max\_squared\_sum, warm\_start\_sag,

is\_saga=(solver == 'saga'))

else:

raise ValueError("solver must be one of {'liblinear', 'lbfgs', "

"'newton-cg', 'sag'}, got '%s' instead" % solver)

if multi\_class == 'multinomial':

n\_classes = max(2, classes.size)

multi\_w0 = np.reshape(w0, (n\_classes, -1))

if n\_classes == 2:

multi\_w0 = multi\_w0[1][np.newaxis, :]

coefs.append(multi\_w0.copy())

else:

coefs.append(w0.copy())

n\_iter[i] = n\_iter\_i

return np.array(coefs), np.array(Cs), n\_iter

# helper function for LogisticCV

def \_log\_reg\_scoring\_path(X, y, train, test, pos\_class=None, Cs=10,

scoring=None, fit\_intercept=False,

max\_iter=100, tol=1e-4, class\_weight=None,

verbose=0, solver='lbfgs', penalty='l2',

dual=False, intercept\_scaling=1.,

multi\_class='warn', random\_state=None,

max\_squared\_sum=None, sample\_weight=None):

"""Computes scores across logistic\_regression\_path

Parameters

----------

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training data.

y : array-like, shape (n\_samples,) or (n\_samples, n\_targets)

Target labels.

train : list of indices

The indices of the train set.

test : list of indices

The indices of the test set.

pos\_class : int, None

The class with respect to which we perform a one-vs-all fit.

If None, then it is assumed that the given problem is binary.

Cs : list of floats | int

Each of the values in Cs describes the inverse of

regularization strength. If Cs is as an int, then a grid of Cs

values are chosen in a logarithmic scale between 1e-4 and 1e4.

If not provided, then a fixed set of values for Cs are used.

scoring : callable or None, optional, default: None

A string (see model evaluation documentation) or

a scorer callable object / function with signature

``scorer(estimator, X, y)``. For a list of scoring functions

that can be used, look at :mod:`sklearn.metrics`. The

default scoring option used is accuracy\_score.

fit\_intercept : bool

If False, then the bias term is set to zero. Else the last

term of each coef\_ gives us the intercept.

max\_iter : int

Maximum number of iterations for the solver.

tol : float

Tolerance for stopping criteria.

class\_weight : dict or 'balanced', optional

Weights associated with classes in the form ``{class\_label: weight}``.

If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust

weights inversely proportional to class frequencies in the input data

as ``n\_samples / (n\_classes \* np.bincount(y))``

Note that these weights will be multiplied with sample\_weight (passed

through the fit method) if sample\_weight is specified.

verbose : int

For the liblinear and lbfgs solvers set verbose to any positive

number for verbosity.

solver : {'lbfgs', 'newton-cg', 'liblinear', 'sag', 'saga'}

Decides which solver to use.

penalty : str, 'l1' or 'l2'

Used to specify the norm used in the penalization. The 'newton-cg',

'sag' and 'lbfgs' solvers support only l2 penalties.

dual : bool

Dual or primal formulation. Dual formulation is only implemented for

l2 penalty with liblinear solver. Prefer dual=False when

n\_samples > n\_features.

intercept\_scaling : float, default 1.

Useful only when the solver 'liblinear' is used

and self.fit\_intercept is set to True. In this case, x becomes

[x, self.intercept\_scaling],

i.e. a "synthetic" feature with constant value equals to

intercept\_scaling is appended to the instance vector.

The intercept becomes intercept\_scaling \* synthetic feature weight

Note! the synthetic feature weight is subject to l1/l2 regularization

as all other features.

To lessen the effect of regularization on synthetic feature weight

(and therefore on the intercept) intercept\_scaling has to be increased.

multi\_class : str, {'ovr', 'multinomial'}

If the option chosen is 'ovr', then a binary problem is fit for each

label. For 'multinomial' the loss minimised is the multinomial loss fit

across the entire probability distribution, \*even when the data is

binary\*. 'multinomial' is unavailable when solver='liblinear'.

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' and

'liblinear'.

max\_squared\_sum : float, default None

Maximum squared sum of X over samples. Used only in SAG solver.

If None, it will be computed, going through all the samples.

The value should be precomputed to speed up cross validation.

sample\_weight : array-like, shape(n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

coefs : ndarray, shape (n\_cs, n\_features) or (n\_cs, n\_features + 1)

List of coefficients for the Logistic Regression model. If

fit\_intercept is set to True then the second dimension will be

n\_features + 1, where the last item represents the intercept.

Cs : ndarray

Grid of Cs used for cross-validation.

scores : ndarray, shape (n\_cs,)

Scores obtained for each Cs.

n\_iter : array, shape(n\_cs,)

Actual number of iteration for each Cs.

"""

X\_train = X[train]

X\_test = X[test]

y\_train = y[train]

y\_test = y[test]

if sample\_weight is not None:

sample\_weight = check\_array(sample\_weight, ensure\_2d=False)

check\_consistent\_length(y, sample\_weight)

sample\_weight = sample\_weight[train]

coefs, Cs, n\_iter = logistic\_regression\_path(

X\_train, y\_train, Cs=Cs, fit\_intercept=fit\_intercept,

solver=solver, max\_iter=max\_iter, class\_weight=class\_weight,

pos\_class=pos\_class, multi\_class=multi\_class,

tol=tol, verbose=verbose, dual=dual, penalty=penalty,

intercept\_scaling=intercept\_scaling, random\_state=random\_state,

check\_input=False, max\_squared\_sum=max\_squared\_sum,

sample\_weight=sample\_weight)

log\_reg = LogisticRegression(solver=solver, multi\_class=multi\_class)

# The score method of Logistic Regression has a classes\_ attribute.

if multi\_class == 'ovr':

log\_reg.classes\_ = np.array([-1, 1])

elif multi\_class == 'multinomial':

log\_reg.classes\_ = np.unique(y\_train)

else:

raise ValueError("multi\_class should be either multinomial or ovr, "

"got %d" % multi\_class)

if pos\_class is not None:

mask = (y\_test == pos\_class)

y\_test = np.ones(y\_test.shape, dtype=np.float64)

y\_test[~mask] = -1.

scores = list()

if isinstance(scoring, six.string\_types):

scoring = get\_scorer(scoring)

for w in coefs:

if multi\_class == 'ovr':

w = w[np.newaxis, :]

if fit\_intercept:

log\_reg.coef\_ = w[:, :-1]

log\_reg.intercept\_ = w[:, -1]

else:

log\_reg.coef\_ = w

log\_reg.intercept\_ = 0.

if scoring is None:

scores.append(log\_reg.score(X\_test, y\_test))

else:

scores.append(scoring(log\_reg, X\_test, y\_test))

return coefs, Cs, np.array(scores), n\_iter

class LogisticRegression(BaseEstimator, LinearClassifierMixin,

SparseCoefMixin):

"""Logistic Regression (aka logit, MaxEnt) classifier.

In the multiclass case, the training algorithm uses the one-vs-rest (OvR)

scheme if the 'multi\_class' option is set to 'ovr', and uses the cross-

entropy loss if the 'multi\_class' option is set to 'multinomial'.

(Currently the 'multinomial' option is supported only by the 'lbfgs',

'sag' and 'newton-cg' solvers.)

This class implements regularized logistic regression using the

'liblinear' library, 'newton-cg', 'sag' and 'lbfgs' solvers. It can handle

both dense and sparse input. Use C-ordered arrays or CSR matrices

containing 64-bit floats for optimal performance; any other input format

will be converted (and copied).

The 'newton-cg', 'sag', and 'lbfgs' solvers support only L2 regularization

with primal formulation. The 'liblinear' solver supports both L1 and L2

regularization, with a dual formulation only for the L2 penalty.

Read more in the :ref:`User Guide <logistic\_regression>`.

Parameters

----------

penalty : str, 'l1' or 'l2', default: 'l2'

Used to specify the norm used in the penalization. The 'newton-cg',

'sag' and 'lbfgs' solvers support only l2 penalties.

.. versionadded:: 0.19

l1 penalty with SAGA solver (allowing 'multinomial' + L1)

dual : bool, default: False

Dual or primal formulation. Dual formulation is only implemented for

l2 penalty with liblinear solver. Prefer dual=False when

n\_samples > n\_features.

tol : float, default: 1e-4

Tolerance for stopping criteria.

C : float, default: 1.0

Inverse of regularization strength; must be a positive float.

Like in support vector machines, smaller values specify stronger

regularization.

fit\_intercept : bool, default: True

Specifies if a constant (a.k.a. bias or intercept) should be

added to the decision function.

intercept\_scaling : float, default 1.

Useful only when the solver 'liblinear' is used

and self.fit\_intercept is set to True. In this case, x becomes

[x, self.intercept\_scaling],

i.e. a "synthetic" feature with constant value equal to

intercept\_scaling is appended to the instance vector.

The intercept becomes ``intercept\_scaling \* synthetic\_feature\_weight``.

Note! the synthetic feature weight is subject to l1/l2 regularization

as all other features.

To lessen the effect of regularization on synthetic feature weight

(and therefore on the intercept) intercept\_scaling has to be increased.

class\_weight : dict or 'balanced', default: None

Weights associated with classes in the form ``{class\_label: weight}``.

If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust

weights inversely proportional to class frequencies in the input data

as ``n\_samples / (n\_classes \* np.bincount(y))``.

Note that these weights will be multiplied with sample\_weight (passed

through the fit method) if sample\_weight is specified.

.. versionadded:: 0.17

\*class\_weight='balanced'\*

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' or

'liblinear'.

solver : str, {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, \

default: 'liblinear'.

Algorithm to use in the optimization problem.

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and

'saga' are faster for large ones.

- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs'

handle multinomial loss; 'liblinear' is limited to one-versus-rest

schemes.

- 'newton-cg', 'lbfgs' and 'sag' only handle L2 penalty, whereas

'liblinear' and 'saga' handle L1 penalty.

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.

.. versionadded:: 0.17

Stochastic Average Gradient descent solver.

.. versionadded:: 0.19

SAGA solver.

.. versionchanged:: 0.20

Default will change from 'liblinear' to 'lbfgs' in 0.22.

max\_iter : int, default: 100

Useful only for the newton-cg, sag and lbfgs solvers.

Maximum number of iterations taken for the solvers to converge.

multi\_class : str, {'ovr', 'multinomial', 'auto'}, default: 'ovr'

If the option chosen is 'ovr', then a binary problem is fit for each

label. For 'multinomial' the loss minimised is the multinomial loss fit

across the entire probability distribution, \*even when the data is

binary\*. 'multinomial' is unavailable when solver='liblinear'.

'auto' selects 'ovr' if the data is binary, or if solver='liblinear',

and otherwise selects 'multinomial'.

.. versionadded:: 0.18

Stochastic Average Gradient descent solver for 'multinomial' case.

.. versionchanged:: 0.20

Default will change from 'ovr' to 'auto' in 0.22.

verbose : int, default: 0

For the liblinear and lbfgs solvers set verbose to any positive

number for verbosity.

warm\_start : bool, default: False

When set to True, reuse the solution of the previous call to fit as

initialization, otherwise, just erase the previous solution.

Useless for liblinear solver. See :term:`the Glossary <warm\_start>`.

.. versionadded:: 0.17

\*warm\_start\* to support \*lbfgs\*, \*newton-cg\*, \*sag\*, \*saga\* solvers.

n\_jobs : int or None, optional (default=None)

Number of CPU cores used when parallelizing over classes if

multi\_class='ovr'". This parameter is ignored when the ``solver`` is

set to 'liblinear' regardless of whether 'multi\_class' is specified or

not. ``None`` means 1 unless in a :obj:`joblib.parallel\_backend`

context. ``-1`` means using all processors.

See :term:`Glossary <n\_jobs>` for more details.

Attributes

----------

coef\_ : array, shape (1, n\_features) or (n\_classes, n\_features)

Coefficient of the features in the decision function.

`coef\_` is of shape (1, n\_features) when the given problem is binary.

In particular, when `multi\_class='multinomial'`, `coef\_` corresponds

to outcome 1 (True) and `-coef\_` corresponds to outcome 0 (False).

intercept\_ : array, shape (1,) or (n\_classes,)

Intercept (a.k.a. bias) added to the decision function.

If `fit\_intercept` is set to False, the intercept is set to zero.

`intercept\_` is of shape (1,) when the given problem is binary.

In particular, when `multi\_class='multinomial'`, `intercept\_`

corresponds to outcome 1 (True) and `-intercept\_` corresponds to

outcome 0 (False).

n\_iter\_ : array, shape (n\_classes,) or (1, )

Actual number of iterations for all classes. If binary or multinomial,

it returns only 1 element. For liblinear solver, only the maximum

number of iteration across all classes is given.

.. versionchanged:: 0.20

In SciPy <= 1.0.0 the number of lbfgs iterations may exceed

``max\_iter``. ``n\_iter\_`` will now report at most ``max\_iter``.

Examples

--------

>>> from sklearn.datasets import load\_iris

>>> from sklearn.linear\_model import LogisticRegression

>>> X, y = load\_iris(return\_X\_y=True)

>>> clf = LogisticRegression(random\_state=0, solver='lbfgs',

... multi\_class='multinomial').fit(X, y)

>>> clf.predict(X[:2, :])

array([0, 0])

>>> clf.predict\_proba(X[:2, :]) # doctest: +ELLIPSIS

array([[9.8...e-01, 1.8...e-02, 1.4...e-08],

[9.7...e-01, 2.8...e-02, ...e-08]])

>>> clf.score(X, y)

0.97...

See also

--------

SGDClassifier : incrementally trained logistic regression (when given

the parameter ``loss="log"``).

LogisticRegressionCV : Logistic regression with built-in cross validation

Notes

-----

The underlying C implementation uses a random number generator to

select features when fitting the model. It is thus not uncommon,

to have slightly different results for the same input data. If

that happens, try with a smaller tol parameter.

Predict output may not match that of standalone liblinear in certain

cases. See :ref:`differences from liblinear <liblinear\_differences>`

in the narrative documentation.

References

----------

LIBLINEAR -- A Library for Large Linear Classification

https://www.csie.ntu.edu.tw/~cjlin/liblinear/

SAG -- Mark Schmidt, Nicolas Le Roux, and Francis Bach

Minimizing Finite Sums with the Stochastic Average Gradient

https://hal.inria.fr/hal-00860051/document

SAGA -- Defazio, A., Bach F. & Lacoste-Julien S. (2014).

SAGA: A Fast Incremental Gradient Method With Support

for Non-Strongly Convex Composite Objectives

https://arxiv.org/abs/1407.0202

Hsiang-Fu Yu, Fang-Lan Huang, Chih-Jen Lin (2011). Dual coordinate descent

methods for logistic regression and maximum entropy models.

Machine Learning 85(1-2):41-75.

https://www.csie.ntu.edu.tw/~cjlin/papers/maxent\_dual.pdf

"""

def \_\_init\_\_(self, penalty='l2', dual=False, tol=1e-4, C=1.0,

fit\_intercept=True, intercept\_scaling=1, class\_weight=None,

random\_state=None, solver='warn', max\_iter=100,

multi\_class='warn', verbose=0, warm\_start=False, n\_jobs=None):

self.penalty = penalty

self.dual = dual

self.tol = tol

self.C = C

self.fit\_intercept = fit\_intercept

self.intercept\_scaling = intercept\_scaling

self.class\_weight = class\_weight

self.random\_state = random\_state

self.solver = solver

self.max\_iter = max\_iter

self.multi\_class = multi\_class

self.verbose = verbose

self.warm\_start = warm\_start

self.n\_jobs = n\_jobs

def fit(self, X, y, sample\_weight=None):

"""Fit the model according to the given training data.

Parameters

----------

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training vector, where n\_samples is the number of samples and

n\_features is the number of features.

y : array-like, shape (n\_samples,)

Target vector relative to X.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

.. versionadded:: 0.17

\*sample\_weight\* support to LogisticRegression.

Returns

-------

self : object

"""

if not isinstance(self.C, numbers.Number) or self.C < 0:

raise ValueError("Penalty term must be positive; got (C=%r)"

% self.C)

if not isinstance(self.max\_iter, numbers.Number) or self.max\_iter < 0:

raise ValueError("Maximum number of iteration must be positive;"

" got (max\_iter=%r)" % self.max\_iter)

if not isinstance(self.tol, numbers.Number) or self.tol < 0:

raise ValueError("Tolerance for stopping criteria must be "

"positive; got (tol=%r)" % self.tol)

solver = \_check\_solver(self.solver, self.penalty, self.dual)

if solver in ['newton-cg']:

\_dtype = [np.float64, np.float32]

else:

\_dtype = np.float64

X, y = check\_X\_y(X, y, accept\_sparse='csr', dtype=\_dtype, order="C",

accept\_large\_sparse=solver != 'liblinear')

check\_classification\_targets(y)

self.classes\_ = np.unique(y)

n\_samples, n\_features = X.shape

multi\_class = \_check\_multi\_class(self.multi\_class, solver,

len(self.classes\_))

if solver == 'liblinear':

if effective\_n\_jobs(self.n\_jobs) != 1:

warnings.warn("'n\_jobs' > 1 does not have any effect when"

" 'solver' is set to 'liblinear'. Got 'n\_jobs'"

" = {}.".format(effective\_n\_jobs(self.n\_jobs)))

self.coef\_, self.intercept\_, n\_iter\_ = \_fit\_liblinear(

X, y, self.C, self.fit\_intercept, self.intercept\_scaling,

self.class\_weight, self.penalty, self.dual, self.verbose,

self.max\_iter, self.tol, self.random\_state,

sample\_weight=sample\_weight)

self.n\_iter\_ = np.array([n\_iter\_])

return self

if solver in ['sag', 'saga']:

max\_squared\_sum = row\_norms(X, squared=True).max()

else:

max\_squared\_sum = None

n\_classes = len(self.classes\_)

classes\_ = self.classes\_

if n\_classes < 2:

raise ValueError("This solver needs samples of at least 2 classes"

" in the data, but the data contains only one"

" class: %r" % classes\_[0])

if len(self.classes\_) == 2:

n\_classes = 1

classes\_ = classes\_[1:]

if self.warm\_start:

warm\_start\_coef = getattr(self, 'coef\_', None)

else:

warm\_start\_coef = None

if warm\_start\_coef is not None and self.fit\_intercept:

warm\_start\_coef = np.append(warm\_start\_coef,

self.intercept\_[:, np.newaxis],

axis=1)

self.coef\_ = list()

self.intercept\_ = np.zeros(n\_classes)

# Hack so that we iterate only once for the multinomial case.

if multi\_class == 'multinomial':

classes\_ = [None]

warm\_start\_coef = [warm\_start\_coef]

if warm\_start\_coef is None:

warm\_start\_coef = [None] \* n\_classes

path\_func = delayed(logistic\_regression\_path)

# The SAG solver releases the GIL so it's more efficient to use

# threads for this solver.

if solver in ['sag', 'saga']:

prefer = 'threads'

else:

prefer = 'processes'

fold\_coefs\_ = Parallel(n\_jobs=self.n\_jobs, verbose=self.verbose,

\*\*\_joblib\_parallel\_args(prefer=prefer))(

path\_func(X, y, pos\_class=class\_, Cs=[self.C],

fit\_intercept=self.fit\_intercept, tol=self.tol,

verbose=self.verbose, solver=solver,

multi\_class=multi\_class, max\_iter=self.max\_iter,

class\_weight=self.class\_weight, check\_input=False,

random\_state=self.random\_state, coef=warm\_start\_coef\_,

penalty=self.penalty,

max\_squared\_sum=max\_squared\_sum,

sample\_weight=sample\_weight)

for class\_, warm\_start\_coef\_ in zip(classes\_, warm\_start\_coef))

fold\_coefs\_, \_, n\_iter\_ = zip(\*fold\_coefs\_)

self.n\_iter\_ = np.asarray(n\_iter\_, dtype=np.int32)[:, 0]

if multi\_class == 'multinomial':

self.coef\_ = fold\_coefs\_[0][0]

else:

self.coef\_ = np.asarray(fold\_coefs\_)

self.coef\_ = self.coef\_.reshape(n\_classes, n\_features +

int(self.fit\_intercept))

if self.fit\_intercept:

self.intercept\_ = self.coef\_[:, -1]

self.coef\_ = self.coef\_[:, :-1]

return self

def predict\_proba(self, X):

"""Probability estimates.

The returned estimates for all classes are ordered by the

label of classes.

For a multi\_class problem, if multi\_class is set to be "multinomial"

the softmax function is used to find the predicted probability of

each class.

Else use a one-vs-rest approach, i.e calculate the probability

of each class assuming it to be positive using the logistic function.

and normalize these values across all the classes.

Parameters

----------

X : array-like, shape = [n\_samples, n\_features]

Returns

-------

T : array-like, shape = [n\_samples, n\_classes]

Returns the probability of the sample for each class in the model,

where classes are ordered as they are in ``self.classes\_``.

"""

if not hasattr(self, "coef\_"):

raise NotFittedError("Call fit before prediction")

ovr = (self.multi\_class in ["ovr", "warn"] or

(self.multi\_class == 'auto' and (self.classes\_.size <= 2 or

self.solver == 'liblinear')))

if ovr:

return super(LogisticRegression, self).\_predict\_proba\_lr(X)

else:

decision = self.decision\_function(X)

if decision.ndim == 1:

# Workaround for multi\_class="multinomial" and binary outcomes

# which requires softmax prediction with only a 1D decision.

decision\_2d = np.c\_[-decision, decision]

else:

decision\_2d = decision

return softmax(decision\_2d, copy=False)

def predict\_log\_proba(self, X):

"""Log of probability estimates.

The returned estimates for all classes are ordered by the

label of classes.

Parameters

----------

X : array-like, shape = [n\_samples, n\_features]

Returns

-------

T : array-like, shape = [n\_samples, n\_classes]

Returns the log-probability of the sample for each class in the

model, where classes are ordered as they are in ``self.classes\_``.

"""

return np.log(self.predict\_proba(X))

class LogisticRegressionCV(LogisticRegression, BaseEstimator,

LinearClassifierMixin):

"""Logistic Regression CV (aka logit, MaxEnt) classifier.

See glossary entry for :term:`cross-validation estimator`.

This class implements logistic regression using liblinear, newton-cg, sag

of lbfgs optimizer. The newton-cg, sag and lbfgs solvers support only L2

regularization with primal formulation. The liblinear solver supports both

L1 and L2 regularization, with a dual formulation only for the L2 penalty.

For the grid of Cs values (that are set by default to be ten values in

a logarithmic scale between 1e-4 and 1e4), the best hyperparameter is

selected by the cross-validator StratifiedKFold, but it can be changed

using the cv parameter. In the case of newton-cg and lbfgs solvers,

we warm start along the path i.e guess the initial coefficients of the

present fit to be the coefficients got after convergence in the previous

fit, so it is supposed to be faster for high-dimensional dense data.

For a multiclass problem, the hyperparameters for each class are computed

using the best scores got by doing a one-vs-rest in parallel across all

folds and classes. Hence this is not the true multinomial loss.

Read more in the :ref:`User Guide <logistic\_regression>`.

Parameters

----------

Cs : list of floats | int

Each of the values in Cs describes the inverse of regularization

strength. If Cs is as an int, then a grid of Cs values are chosen

in a logarithmic scale between 1e-4 and 1e4.

Like in support vector machines, smaller values specify stronger

regularization.

fit\_intercept : bool, default: True

Specifies if a constant (a.k.a. bias or intercept) should be

added to the decision function.

cv : integer or cross-validation generator, default: None

The default cross-validation generator used is Stratified K-Folds.

If an integer is provided, then it is the number of folds used.

See the module :mod:`sklearn.model\_selection` module for the

list of possible cross-validation objects.

.. versionchanged:: 0.20

``cv`` default value if None will change from 3-fold to 5-fold

in v0.22.

dual : bool

Dual or primal formulation. Dual formulation is only implemented for

l2 penalty with liblinear solver. Prefer dual=False when

n\_samples > n\_features.

penalty : str, 'l1' or 'l2'

Used to specify the norm used in the penalization. The 'newton-cg',

'sag' and 'lbfgs' solvers support only l2 penalties.

scoring : string, callable, or None

A string (see model evaluation documentation) or

a scorer callable object / function with signature

``scorer(estimator, X, y)``. For a list of scoring functions

that can be used, look at :mod:`sklearn.metrics`. The

default scoring option used is 'accuracy'.

solver : str, {'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, \

default: 'lbfgs'.

Algorithm to use in the optimization problem.

- For small datasets, 'liblinear' is a good choice, whereas 'sag' and

'saga' are faster for large ones.

- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs'

handle multinomial loss; 'liblinear' is limited to one-versus-rest

schemes.

- 'newton-cg', 'lbfgs' and 'sag' only handle L2 penalty, whereas

'liblinear' and 'saga' handle L1 penalty.

- 'liblinear' might be slower in LogisticRegressionCV because it does

not handle warm-starting.

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.

.. versionadded:: 0.17

Stochastic Average Gradient descent solver.

.. versionadded:: 0.19

SAGA solver.

tol : float, optional

Tolerance for stopping criteria.

max\_iter : int, optional

Maximum number of iterations of the optimization algorithm.

class\_weight : dict or 'balanced', optional

Weights associated with classes in the form ``{class\_label: weight}``.

If not given, all classes are supposed to have weight one.

The "balanced" mode uses the values of y to automatically adjust

weights inversely proportional to class frequencies in the input data

as ``n\_samples / (n\_classes \* np.bincount(y))``.

Note that these weights will be multiplied with sample\_weight (passed

through the fit method) if sample\_weight is specified.

.. versionadded:: 0.17

class\_weight == 'balanced'

n\_jobs : int or None, optional (default=None)

Number of CPU cores used during the cross-validation loop.

``None`` means 1 unless in a :obj:`joblib.parallel\_backend` context.

``-1`` means using all processors. See :term:`Glossary <n\_jobs>`

for more details.

verbose : int

For the 'liblinear', 'sag' and 'lbfgs' solvers set verbose to any

positive number for verbosity.

refit : bool

If set to True, the scores are averaged across all folds, and the

coefs and the C that corresponds to the best score is taken, and a

final refit is done using these parameters.

Otherwise the coefs, intercepts and C that correspond to the

best scores across folds are averaged.

intercept\_scaling : float, default 1.

Useful only when the solver 'liblinear' is used

and self.fit\_intercept is set to True. In this case, x becomes

[x, self.intercept\_scaling],

i.e. a "synthetic" feature with constant value equal to

intercept\_scaling is appended to the instance vector.

The intercept becomes ``intercept\_scaling \* synthetic\_feature\_weight``.

Note! the synthetic feature weight is subject to l1/l2 regularization

as all other features.

To lessen the effect of regularization on synthetic feature weight

(and therefore on the intercept) intercept\_scaling has to be increased.

multi\_class : str, {'ovr', 'multinomial', 'auto'}, default: 'ovr'

If the option chosen is 'ovr', then a binary problem is fit for each

label. For 'multinomial' the loss minimised is the multinomial loss fit

across the entire probability distribution, \*even when the data is

binary\*. 'multinomial' is unavailable when solver='liblinear'.

'auto' selects 'ovr' if the data is binary, or if solver='liblinear',

and otherwise selects 'multinomial'.

.. versionadded:: 0.18

Stochastic Average Gradient descent solver for 'multinomial' case.

.. versionchanged:: 0.20

Default will change from 'ovr' to 'auto' in 0.22.

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`.

Attributes

----------

coef\_ : array, shape (1, n\_features) or (n\_classes, n\_features)

Coefficient of the features in the decision function.

`coef\_` is of shape (1, n\_features) when the given problem

is binary.

intercept\_ : array, shape (1,) or (n\_classes,)

Intercept (a.k.a. bias) added to the decision function.

If `fit\_intercept` is set to False, the intercept is set to zero.

`intercept\_` is of shape(1,) when the problem is binary.

Cs\_ : array

Array of C i.e. inverse of regularization parameter values used

for cross-validation.

coefs\_paths\_ : array, shape ``(n\_folds, len(Cs\_), n\_features)`` or \

``(n\_folds, len(Cs\_), n\_features + 1)``

dict with classes as the keys, and the path of coefficients obtained

during cross-validating across each fold and then across each Cs

after doing an OvR for the corresponding class as values.

If the 'multi\_class' option is set to 'multinomial', then

the coefs\_paths are the coefficients corresponding to each class.

Each dict value has shape ``(n\_folds, len(Cs\_), n\_features)`` or

``(n\_folds, len(Cs\_), n\_features + 1)`` depending on whether the

intercept is fit or not.

scores\_ : dict

dict with classes as the keys, and the values as the

grid of scores obtained during cross-validating each fold, after doing

an OvR for the corresponding class. If the 'multi\_class' option

given is 'multinomial' then the same scores are repeated across

all classes, since this is the multinomial class.

Each dict value has shape (n\_folds, len(Cs))

C\_ : array, shape (n\_classes,) or (n\_classes - 1,)

Array of C that maps to the best scores across every class. If refit is

set to False, then for each class, the best C is the average of the

C's that correspond to the best scores for each fold.

`C\_` is of shape(n\_classes,) when the problem is binary.

n\_iter\_ : array, shape (n\_classes, n\_folds, n\_cs) or (1, n\_folds, n\_cs)

Actual number of iterations for all classes, folds and Cs.

In the binary or multinomial cases, the first dimension is equal to 1.

Examples

--------

>>> from sklearn.datasets import load\_iris

>>> from sklearn.linear\_model import LogisticRegressionCV

>>> X, y = load\_iris(return\_X\_y=True)

>>> clf = LogisticRegressionCV(cv=5, random\_state=0,

... multi\_class='multinomial').fit(X, y)

>>> clf.predict(X[:2, :])

array([0, 0])

>>> clf.predict\_proba(X[:2, :]).shape

(2, 3)

>>> clf.score(X, y) # doctest: +ELLIPSIS

0.98...

See also

--------

LogisticRegression

"""

def \_\_init\_\_(self, Cs=10, fit\_intercept=True, cv='warn', dual=False,

penalty='l2', scoring=None, solver='lbfgs', tol=1e-4,

max\_iter=100, class\_weight=None, n\_jobs=None, verbose=0,

refit=True, intercept\_scaling=1., multi\_class='warn',

random\_state=None):

self.Cs = Cs

self.fit\_intercept = fit\_intercept

self.cv = cv

self.dual = dual

self.penalty = penalty

self.scoring = scoring

self.tol = tol

self.max\_iter = max\_iter

self.class\_weight = class\_weight

self.n\_jobs = n\_jobs

self.verbose = verbose

self.solver = solver

self.refit = refit

self.intercept\_scaling = intercept\_scaling

self.multi\_class = multi\_class

self.random\_state = random\_state

def fit(self, X, y, sample\_weight=None):

"""Fit the model according to the given training data.

Parameters

----------

X : {array-like, sparse matrix}, shape (n\_samples, n\_features)

Training vector, where n\_samples is the number of samples and

n\_features is the number of features.

y : array-like, shape (n\_samples,)

Target vector relative to X.

sample\_weight : array-like, shape (n\_samples,) optional

Array of weights that are assigned to individual samples.

If not provided, then each sample is given unit weight.

Returns

-------

self : object

"""

solver = \_check\_solver(self.solver, self.penalty, self.dual)

if not isinstance(self.max\_iter, numbers.Number) or self.max\_iter < 0:

raise ValueError("Maximum number of iteration must be positive;"

" got (max\_iter=%r)" % self.max\_iter)

if not isinstance(self.tol, numbers.Number) or self.tol < 0:

raise ValueError("Tolerance for stopping criteria must be "

"positive; got (tol=%r)" % self.tol)

X, y = check\_X\_y(X, y, accept\_sparse='csr', dtype=np.float64,

order="C",

accept\_large\_sparse=solver != 'liblinear')

check\_classification\_targets(y)

class\_weight = self.class\_weight

# Encode for string labels

label\_encoder = LabelEncoder().fit(y)

y = label\_encoder.transform(y)

if isinstance(class\_weight, dict):

class\_weight = dict((label\_encoder.transform([cls])[0], v)

for cls, v in class\_weight.items())

# The original class labels

classes = self.classes\_ = label\_encoder.classes\_

encoded\_labels = label\_encoder.transform(label\_encoder.classes\_)

multi\_class = \_check\_multi\_class(self.multi\_class, solver,

len(classes))

if solver in ['sag', 'saga']:

max\_squared\_sum = row\_norms(X, squared=True).max()

else:

max\_squared\_sum = None

# init cross-validation generator

cv = check\_cv(self.cv, y, classifier=True)

folds = list(cv.split(X, y))

# Use the label encoded classes

n\_classes = len(encoded\_labels)

if n\_classes < 2:

raise ValueError("This solver needs samples of at least 2 classes"

" in the data, but the data contains only one"

" class: %r" % classes[0])

if n\_classes == 2:

# OvR in case of binary problems is as good as fitting

# the higher label

n\_classes = 1

encoded\_labels = encoded\_labels[1:]

classes = classes[1:]

# We need this hack to iterate only once over labels, in the case of

# multi\_class = multinomial, without changing the value of the labels.

if multi\_class == 'multinomial':

iter\_encoded\_labels = iter\_classes = [None]

else:

iter\_encoded\_labels = encoded\_labels

iter\_classes = classes

# compute the class weights for the entire dataset y

if class\_weight == "balanced":

class\_weight = compute\_class\_weight(class\_weight,

np.arange(len(self.classes\_)),

y)

class\_weight = dict(enumerate(class\_weight))

path\_func = delayed(\_log\_reg\_scoring\_path)

# The SAG solver releases the GIL so it's more efficient to use

# threads for this solver.

if self.solver in ['sag', 'saga']:

prefer = 'threads'

else:

prefer = 'processes'

fold\_coefs\_ = Parallel(n\_jobs=self.n\_jobs, verbose=self.verbose,

\*\*\_joblib\_parallel\_args(prefer=prefer))(

path\_func(X, y, train, test, pos\_class=label, Cs=self.Cs,

fit\_intercept=self.fit\_intercept, penalty=self.penalty,

dual=self.dual, solver=solver, tol=self.tol,

max\_iter=self.max\_iter, verbose=self.verbose,

class\_weight=class\_weight, scoring=self.scoring,

multi\_class=multi\_class,

intercept\_scaling=self.intercept\_scaling,

random\_state=self.random\_state,

max\_squared\_sum=max\_squared\_sum,

sample\_weight=sample\_weight

)

for label in iter\_encoded\_labels

for train, test in folds)

if multi\_class == 'multinomial':

multi\_coefs\_paths, Cs, multi\_scores, n\_iter\_ = zip(\*fold\_coefs\_)

multi\_coefs\_paths = np.asarray(multi\_coefs\_paths)

multi\_scores = np.asarray(multi\_scores)

# This is just to maintain API similarity between the ovr and

# multinomial option.

# Coefs\_paths in now n\_folds X len(Cs) X n\_classes X n\_features

# we need it to be n\_classes X len(Cs) X n\_folds X n\_features

# to be similar to "ovr".

coefs\_paths = np.rollaxis(multi\_coefs\_paths, 2, 0)

# Multinomial has a true score across all labels. Hence the

# shape is n\_folds X len(Cs). We need to repeat this score

# across all labels for API similarity.

scores = np.tile(multi\_scores, (n\_classes, 1, 1))

self.Cs\_ = Cs[0]

self.n\_iter\_ = np.reshape(n\_iter\_, (1, len(folds),

len(self.Cs\_)))

else:

coefs\_paths, Cs, scores, n\_iter\_ = zip(\*fold\_coefs\_)

self.Cs\_ = Cs[0]

coefs\_paths = np.reshape(coefs\_paths, (n\_classes, len(folds),

len(self.Cs\_), -1))

self.n\_iter\_ = np.reshape(n\_iter\_, (n\_classes, len(folds),

len(self.Cs\_)))

self.coefs\_paths\_ = dict(zip(classes, coefs\_paths))

scores = np.reshape(scores, (n\_classes, len(folds), -1))

self.scores\_ = dict(zip(classes, scores))

self.C\_ = list()

self.coef\_ = np.empty((n\_classes, X.shape[1]))

self.intercept\_ = np.zeros(n\_classes)

# hack to iterate only once for multinomial case.

if multi\_class == 'multinomial':

scores = multi\_scores

coefs\_paths = multi\_coefs\_paths

for index, (cls, encoded\_label) in enumerate(

zip(iter\_classes, iter\_encoded\_labels)):

if multi\_class == 'ovr':

# The scores\_ / coefs\_paths\_ dict have unencoded class

# labels as their keys

scores = self.scores\_[cls]

coefs\_paths = self.coefs\_paths\_[cls]

if self.refit:

best\_index = scores.sum(axis=0).argmax()

C\_ = self.Cs\_[best\_index]

self.C\_.append(C\_)

if multi\_class == 'multinomial':

coef\_init = np.mean(coefs\_paths[:, best\_index, :, :],

axis=0)

else:

coef\_init = np.mean(coefs\_paths[:, best\_index, :], axis=0)

# Note that y is label encoded and hence pos\_class must be

# the encoded label / None (for 'multinomial')

w, \_, \_ = logistic\_regression\_path(

X, y, pos\_class=encoded\_label, Cs=[C\_], solver=solver,

fit\_intercept=self.fit\_intercept, coef=coef\_init,

max\_iter=self.max\_iter, tol=self.tol,

penalty=self.penalty,

class\_weight=class\_weight,

multi\_class=multi\_class,

verbose=max(0, self.verbose - 1),

random\_state=self.random\_state,

check\_input=False, max\_squared\_sum=max\_squared\_sum,

sample\_weight=sample\_weight)

w = w[0]

else:

# Take the best scores across every fold and the average of all

# coefficients corresponding to the best scores.

best\_indices = np.argmax(scores, axis=1)

w = np.mean([coefs\_paths[i][best\_indices[i]]

for i in range(len(folds))], axis=0)

self.C\_.append(np.mean(self.Cs\_[best\_indices]))

if multi\_class == 'multinomial':

self.C\_ = np.tile(self.C\_, n\_classes)

self.coef\_ = w[:, :X.shape[1]]

if self.fit\_intercept:

self.intercept\_ = w[:, -1]

else:

self.coef\_[index] = w[: X.shape[1]]

if self.fit\_intercept:

self.intercept\_[index] = w[-1]

self.C\_ = np.asarray(self.C\_)

return self

def score(self, X, y, sample\_weight=None):

"""Returns the score using the `scoring` option on the given

test data and labels.

Parameters

----------

X : array-like, shape = (n\_samples, n\_features)

Test samples.

y : array-like, shape = (n\_samples,)

True labels for X.

sample\_weight : array-like, shape = [n\_samples], optional

Sample weights.

Returns

-------

score : float

Score of self.predict(X) wrt. y.

"""

if self.scoring is not None:

warnings.warn("The long-standing behavior to use the "

"accuracy score has changed. The scoring "

"parameter is now used. "

"This warning will disappear in version 0.22.",

ChangedBehaviorWarning)

scoring = self.scoring or 'accuracy'

if isinstance(scoring, six.string\_types):

scoring = get\_scorer(scoring)

return scoring(self, X, y, sample\_weight=sample\_weight)