Name: Manasi Amdekar
Roll No.: J003

Support Vector Classification API

C-Support Vector Classification.

The implementation is based on libsvm. The fit time scales at least quadratically with the number of samples and may be impractical beyond tens of thousands of samples. For large datasets consider using **LinearSVC** or **SGDClassifier** instead, possibly after a **Nystroem** transformer.

The multiclass support is handled according to a one-vs-one scheme.

Code:
sklearn.svm.**SVC**(***, *C=1.0*, *kernel='rbf'*, *degree=3*, *gamma='scale'*, *coef0=0.0*, *shrinking=True*, *probability=False*, *tol=0.001*, *cache_size=200*, *class_weight=None*, *verbose=False*, *max_iter=-1*, *decision_function_shape='ovr'*, *break_ties=False*, *random_state=None*)

Parameters

C*float, default=1.0*

Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel*{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'}, default='rbf'*

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape (n_samples, n_samples).

degree*int, default=3*

Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

gamma*{'scale', 'auto'} or float, default='scale'*

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'.

- if gamma='scale' (default) is passed then it uses 1 / (n_features * X.var()) as value of gamma,
- if 'auto', uses 1 / n_features.

coef0*float, default=0.0*

Independent term in kernel function. It is only significant in 'poly' and 'sigmoid'.

shrinking*bool, default=True*

Whether to use the shrinking heuristic.

probability*bool, default=False*

Whether to enable probability estimates. This must be enabled prior to calling fit, will slow down that method as it internally uses 5-fold cross-validation, and predict_proba may be inconsistent with predict.

tol*float, default=1e-3*

Tolerance for stopping criterion.

cache_size*float, default=200*

Specify the size of the kernel cache (in MB).

class_weight*dict or 'balanced', default=None*

Set the parameter C of class i to class_weight[i]*C for SVC. 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))

verbose*bool, default=False*

Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

max_iter*int, default=-1*

Hard limit on iterations within solver, or -1 for no limit.

decision_function_shape*{'ovo', 'ovr'}, default='ovr'*

Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, one-vs-one ('ovo') is always used as multi-class strategy. The parameter is ignored for binary classification.

break_ties*bool, default=False*

If true, decision_function_shape='ovr', and number of classes > 2, predict will break ties according to the confidence values of decision_function; otherwise the first class among the tied classes is returned.

random_state*int, RandomState instance or None, default=None*

Controls the pseudo random number generation for shuffling the data for probability estimates. Ignored when probability is False. Pass an int for reproducible output across multiple function calls.

Attributes

class_weight_*ndarray of shape (n_classes,)*

Multipliers of parameter C for each class. Computed based on the class_weight parameter.

classes_*ndarray of shape (n_classes,)*

The classes labels.

coef_*ndarray of shape (n_classes * (n_classes - 1) / 2, n_features)*

Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

coef_ is a readonly property derived from dual_coef_ and support_vectors_.

dual_coef_*ndarray of shape (n_classes -1, n_SV)*

Dual coefficients of the support vector in the decision function, multiplied by their targets. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial.

fit_status_*int*

0 if correctly fitted, 1 otherwise (will raise warning)

intercept_*ndarray of shape (n_classes * (n_classes - 1) / 2,)*

Constants in decision function.

support_*ndarray of shape (n_SV)*

Indices of support vectors.

support_vectors_*ndarray of shape (n_SV, n_features)*

Support vectors.

n_support_*ndarray of shape (n_classes,), dtype=int32*

Number of support vectors for each class.

probA_*ndarray of shape (n_classes * (n_classes - 1) / 2)*

probB_*ndarray of shape (n_classes * (n_classes - 1) / 2)*

If probability=True, it corresponds to the parameters learned in Platt scaling to produce probability estimates from decision values. If probability=False, it's an empty array. Platt scaling uses the logistic function 1 / (1 + exp(decision_value * probA_ + probB_)) where probA_ and probB_ are learned from the dataset.

shape_fit_*tuple of int of shape (n_dimensions_of_X,)*

Array dimensions of training vector X.

How sklearn handles SVM

Support Vector Machines can be employed in both types of classification and regression problems. SVM offers very high accuracy compared to other classifiers such as logistic regression, and decision trees. It can easily handle multiple continuous and categorical variables. SVM constructs a hyperplane in multidimensional space to separate different classes. SVM generates optimal hyperplane in an iterative manner, which is used to minimize an error. The core idea of SVM is to find a maximum marginal hyperplane (MMH) that best divides the dataset into classes.

In python, after we load and explore the data a little bit, we import the SVM module and create a SVC classifier object using the SVC() or LinearSVC() function.

We then fit the model and perform prediction on the test set using the fit() and predict() method respectively.