**1.4. Support Vector Machines**

**Support vector machines (SVMs)** are a set of supervised learning methods used for [classification](http://scikit-learn.org/stable/modules/svm.html#svm-classification), [regression](http://scikit-learn.org/stable/modules/svm.html#svm-regression) and [outliers detection](http://scikit-learn.org/stable/modules/svm.html#svm-outlier-detection).

The advantages of support vector machines are:

* Effective in high dimensional spaces.
* Still effective in cases where number of dimensions is greater than the number of samples.
* Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
* Versatile: different [Kernel functions](http://scikit-learn.org/stable/modules/svm.html#svm-kernels) can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

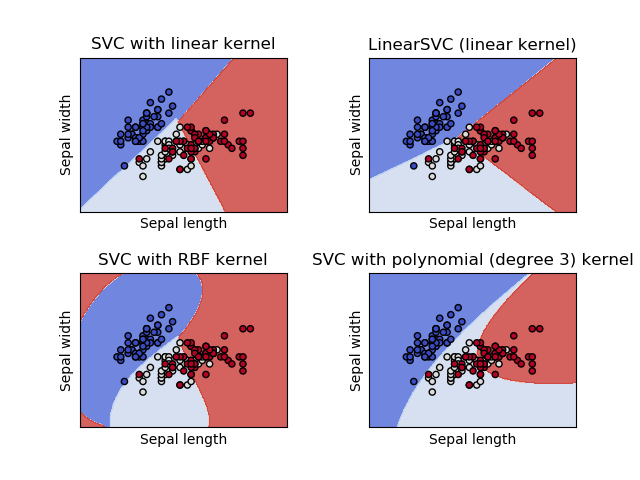
The disadvantages of support vector machines include:

* If the number of features is much greater than the number of samples, avoid over-fitting in choosing [Kernel functions](http://scikit-learn.org/stable/modules/svm.html#svm-kernels) and regularization term is crucial.
* SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see [Scores and probabilities](http://scikit-learn.org/stable/modules/svm.html#scores-probabilities), below).

The support vector machines in scikit-learn support both dense (numpy.ndarray and convertible to that by numpy.asarray) and sparse (any scipy.sparse) sample vectors as input. However, to use an SVM to make predictions for sparse data, it must have been fit on such data. For optimal performance, use C-ordered numpy.ndarray (dense) orscipy.sparse.csr\_matrix (sparse) with dtype=float64.

**1.4.1. Classification**

[**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC) and [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) are classes capable of performing multi-class classification on a dataset.

[](http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html)

[**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) and [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC) are similar methods, but accept slightly different sets of parameters and have different mathematical formulations (see section [Mathematical formulation](http://scikit-learn.org/stable/modules/svm.html#svm-mathematical-formulation)). On the other hand, [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) is another implementation of Support Vector Classification for the case of a linear kernel. Note that [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) does not accept keyword kernel, as this is assumed to be linear. It also lacks some of the members of [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) and [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC), like support\_.

As other classifiers, [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC) and [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) take as input two arrays: an array X of size [n\_samples, n\_features]holding the training samples, and an array y of class labels (strings or integers), size [n\_samples]:

>>>

**>>> from** **sklearn** **import** svm

**>>>** X = [[0, 0], [1, 1]]

**>>>** y = [0, 1]

**>>>** clf = svm.SVC()

**>>>** clf.fit(X, y)

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='auto', kernel='rbf',

max\_iter=-1, probability=False, random\_state=None, shrinking=True,

tol=0.001, verbose=False)

After being fitted, the model can then be used to predict new values:

>>>

**>>>** clf.predict([[2., 2.]])

array([1])

SVMs decision function depends on some subset of the training data, called the support vectors. Some properties of these support vectors can be found in members support\_vectors\_, support\_ and n\_support:

>>>

**>>>** *# get support vectors*

**>>>** clf.support\_vectors\_

array([[ 0., 0.],

[ 1., 1.]])

**>>>** *# get indices of support vectors*

**>>>** clf.support\_

array([0, 1]...)

**>>>** *# get number of support vectors for each class*

**>>>** clf.n\_support\_

array([1, 1]...)

**1.4.1.1. Multi-class classification**

[**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) and [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC) implement the “one-against-one” approach (Knerr et al., 1990) for multi- class classification. If n\_class is the number of classes, then n\_class \* (n\_class - 1) / 2 classifiers are constructed and each one trains data from two classes. To provide a consistent interface with other classifiers, the decision\_function\_shape option allows to aggregate the results of the “one-against-one” classifiers to a decision function of shape (n\_samples, n\_classes):

>>>

**>>>** X = [[0], [1], [2], [3]]

**>>>** Y = [0, 1, 2, 3]

**>>>** clf = svm.SVC(decision\_function\_shape='ovo')

**>>>** clf.fit(X, Y)

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovo', degree=3, gamma='auto', kernel='rbf',

max\_iter=-1, probability=False, random\_state=None, shrinking=True,

tol=0.001, verbose=False)

**>>>** dec = clf.decision\_function([[1]])

**>>>** dec.shape[1] *# 4 classes: 4\*3/2 = 6*

6

**>>>** clf.decision\_function\_shape = "ovr"

**>>>** dec = clf.decision\_function([[1]])

**>>>** dec.shape[1] *# 4 classes*

4

On the other hand, [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) implements “one-vs-the-rest” multi-class strategy, thus training n\_class models. If there are only two classes, only one model is trained:

>>>

**>>>** lin\_clf = svm.LinearSVC()

**>>>** lin\_clf.fit(X, Y)

LinearSVC(C=1.0, class\_weight=None, dual=True, fit\_intercept=True,

intercept\_scaling=1, loss='squared\_hinge', max\_iter=1000,

multi\_class='ovr', penalty='l2', random\_state=None, tol=0.0001,

verbose=0)

**>>>** dec = lin\_clf.decision\_function([[1]])

**>>>** dec.shape[1]

4

See [Mathematical formulation](http://scikit-learn.org/stable/modules/svm.html#svm-mathematical-formulation) for a complete description of the decision function.

Note that the [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) also implements an alternative multi-class strategy, the so-called multi-class SVM formulated by Crammer and Singer, by using the option multi\_class='crammer\_singer'. This method is consistent, which is not true for one-vs-rest classification. In practice, one-vs-rest classification is usually preferred, since the results are mostly similar, but the runtime is significantly less.

For “one-vs-rest” [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) the attributes coef\_ and intercept\_ have the shape [n\_class, n\_features] and [n\_class] respectively. Each row of the coefficients corresponds to one of the n\_class many “one-vs-rest” classifiers and similar for the intercepts, in the order of the “one” class.

In the case of “one-vs-one” [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), the layout of the attributes is a little more involved. In the case of having a linear kernel, The layout of coef\_ and intercept\_ is similar to the one described for [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) described above, except that the shape of coef\_ is [n\_class \* (n\_class - 1) / 2, n\_features], corresponding to as many binary classifiers. The order for classes 0 to n is “0 vs 1”, “0 vs 2” , … “0 vs n”, “1 vs 2”, “1 vs 3”, “1 vs n”, . . . “n-1 vs n”.

The shape of dual\_coef\_ is [n\_class-1, n\_SV] with a somewhat hard to grasp layout. The columns correspond to the support vectors involved in any of the n\_class \* (n\_class - 1) / 2 “one-vs-one” classifiers. Each of the support vectors is used in n\_class - 1 classifiers. The n\_class - 1 entries in each row correspond to the dual coefficients for these classifiers.

This might be made more clear by an example:

Consider a three class problem with class 0 having three support vectors v^{0}_0, v^{1}_0, v^{2}_0 and class 1 and 2 having two support vectors v^{0}_1, v^{1}_1 and v^{0}_2, v^{1}_2 respectively. For each support vector v^{j}_i, there are two dual coefficients. Let’s call the coefficient of support vector v^{j}_i in the classifier between classes i and k \alpha^{j}_{i,k}. Then dual\_coef\_ looks like this:

|  |  |  |
| --- | --- | --- |
| \alpha^{0}_{0,1} | \alpha^{0}_{0,2} | Coefficients for SVs of class 0 |
| \alpha^{1}_{0,1} | \alpha^{1}_{0,2} |
| \alpha^{2}_{0,1} | \alpha^{2}_{0,2} |
| \alpha^{0}_{1,0} | \alpha^{0}_{1,2} | Coefficients for SVs of class 1 |
| \alpha^{1}_{1,0} | \alpha^{1}_{1,2} |
| \alpha^{0}_{2,0} | \alpha^{0}_{2,1} | Coefficients for SVs of class 2 |
| \alpha^{1}_{2,0} | \alpha^{1}_{2,1} |

**1.4.1.2. Scores and probabilities**

The [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) method decision\_function gives per-class scores for each sample (or a single score per sample in the binary case). When the constructor option probability is set to True, class membership probability estimates (from the methods predict\_proba and predict\_log\_proba) are enabled. In the binary case, the probabilities are calibrated using Platt scaling: logistic regression on the SVM’s scores, fit by an additional cross-validation on the training data. In the multiclass case, this is extended as per Wu et al. (2004).

Needless to say, the cross-validation involved in Platt scaling is an expensive operation for large datasets. In addition, the probability estimates may be inconsistent with the scores, in the sense that the “argmax” of the scores may not be the argmax of the probabilities. (E.g., in binary classification, a sample may be labeled by predict as belonging to a class that has probability <½ according to predict\_proba.) Platt’s method is also known to have theoretical issues. If confidence scores are required, but these do not have to be probabilities, then it is advisable to set probability=False and use decision\_function instead of predict\_proba.

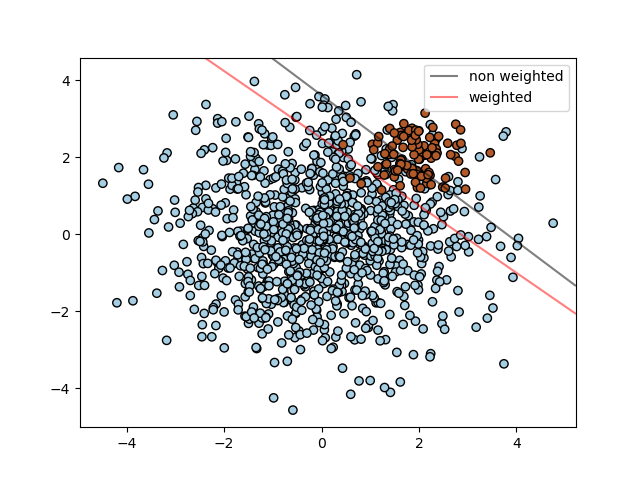
**References:**

* Wu, Lin and Weng, [“Probability estimates for multi-class classification by pairwise coupling”](http://www.csie.ntu.edu.tw/~cjlin/papers/svmprob/svmprob.pdf), JMLR 5:975-1005, 2004.
* Platt “Probabilistic outputs for SVMs and comparisons to regularized likelihood methods” <http://www.cs.colorado.edu/~mozer/Teaching/syllabi/6622/papers/Platt1999.pdf>.

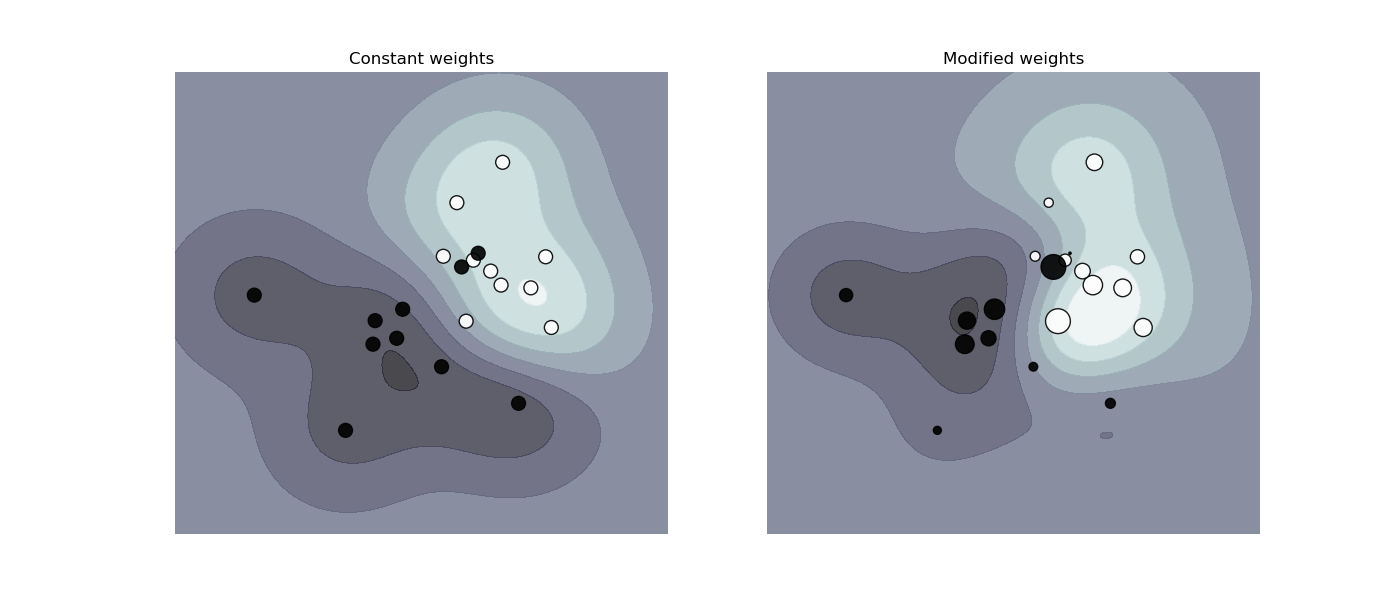
**1.4.1.3. Unbalanced problems**

In problems where it is desired to give more importance to certain classes or certain individual samples keywords class\_weight and sample\_weight can be used.

[**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) (but not [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC)) implement a keyword class\_weight in the fit method. It’s a dictionary of the form{class\_label : value}, where value is a floating point number > 0 that sets the parameter C of class class\_label to C \* value.

[](http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane_unbalanced.html)

[**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC), [**SVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR), [**NuSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html#sklearn.svm.NuSVR) and [**OneClassSVM**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html#sklearn.svm.OneClassSVM) implement also weights for individual samples in method fit through keyword sample\_weight. Similar to class\_weight, these set the parameter C for the i-th example to C \* sample\_weight[i].

[](http://scikit-learn.org/stable/auto_examples/svm/plot_weighted_samples.html)

**Examples:**

* [Plot different SVM classifiers in the iris dataset](http://scikit-learn.org/stable/auto_examples/svm/plot_iris.html#sphx-glr-auto-examples-svm-plot-iris-py),
* [SVM: Maximum margin separating hyperplane](http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html#sphx-glr-auto-examples-svm-plot-separating-hyperplane-py),
* [SVM: Separating hyperplane for unbalanced classes](http://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane_unbalanced.html#sphx-glr-auto-examples-svm-plot-separating-hyperplane-unbalanced-py)
* [SVM-Anova: SVM with univariate feature selection](http://scikit-learn.org/stable/auto_examples/svm/plot_svm_anova.html#sphx-glr-auto-examples-svm-plot-svm-anova-py),
* [Non-linear SVM](http://scikit-learn.org/stable/auto_examples/svm/plot_svm_nonlinear.html#sphx-glr-auto-examples-svm-plot-svm-nonlinear-py)
* [SVM: Weighted samples](http://scikit-learn.org/stable/auto_examples/svm/plot_weighted_samples.html#sphx-glr-auto-examples-svm-plot-weighted-samples-py),

**1.4.2. Regression**

The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression.

The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function for building the model ignores any training data close to the model prediction.

There are three different implementations of Support Vector Regression: [**SVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR), [**NuSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html#sklearn.svm.NuSVR) and [**LinearSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html#sklearn.svm.LinearSVR). [**LinearSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html#sklearn.svm.LinearSVR) provides a faster implementation than [**SVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR) but only considers linear kernels, while [**NuSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html#sklearn.svm.NuSVR) implements a slightly different formulation than [**SVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR) and [**LinearSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVR.html#sklearn.svm.LinearSVR). See [Implementation details](http://scikit-learn.org/stable/modules/svm.html#svm-implementation-details) for further details.

As with classification classes, the fit method will take as argument vectors X, y, only that in this case y is expected to have floating point values instead of integer values:

>>>

**>>> from** **sklearn** **import** svm

**>>>** X = [[0, 0], [2, 2]]

**>>>** y = [0.5, 2.5]

**>>>** clf = svm.SVR()

**>>>** clf.fit(X, y)

SVR(C=1.0, cache\_size=200, coef0=0.0, degree=3, epsilon=0.1, gamma='auto',

kernel='rbf', max\_iter=-1, shrinking=True, tol=0.001, verbose=False)

**>>>** clf.predict([[1, 1]])

array([ 1.5])

**Examples:**

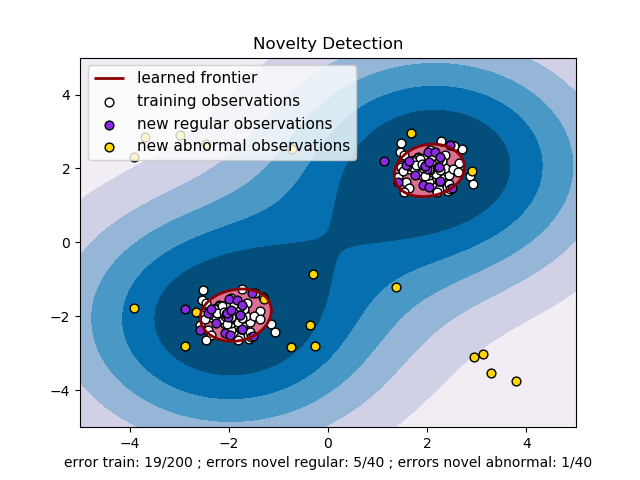
* [Support Vector Regression (SVR) using linear and non-linear kernels](http://scikit-learn.org/stable/auto_examples/svm/plot_svm_regression.html#sphx-glr-auto-examples-svm-plot-svm-regression-py)

**1.4.3. Density estimation, novelty detection**

One-class SVM is used for novelty detection, that is, given a set of samples, it will detect the soft boundary of that set so as to classify new points as belonging to that set or not. The class that implements this is called [**OneClassSVM**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html#sklearn.svm.OneClassSVM).

In this case, as it is a type of unsupervised learning, the fit method will only take as input an array X, as there are no class labels.

See, section [Novelty and Outlier Detection](http://scikit-learn.org/stable/modules/outlier_detection.html#outlier-detection) for more details on this usage.

[](http://scikit-learn.org/stable/auto_examples/svm/plot_oneclass.html)

**Examples:**

* [One-class SVM with non-linear kernel (RBF)](http://scikit-learn.org/stable/auto_examples/svm/plot_oneclass.html#sphx-glr-auto-examples-svm-plot-oneclass-py)
* [Species distribution modeling](http://scikit-learn.org/stable/auto_examples/applications/plot_species_distribution_modeling.html#sphx-glr-auto-examples-applications-plot-species-distribution-modeling-py)

**1.4.4. Complexity**

Support Vector Machines are powerful tools, but their compute and storage requirements increase rapidly with the number of training vectors. The core of an SVM is a quadratic programming problem (QP), separating support vectors from the rest of the training data. The QP solver used by this [libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/)-based implementation scales between O(n_{features} \times n_{samples}^2)and O(n_{features} \times n_{samples}^3) depending on how efficiently the [libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/) cache is used in practice (dataset dependent). If the data is very sparse n_{features} should be replaced by the average number of non-zero features in a sample vector.

Also note that for the linear case, the algorithm used in [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) by the [liblinear](http://www.csie.ntu.edu.tw/~cjlin/liblinear/) implementation is much more efficient than its [libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/)-based [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC) counterpart and can scale almost linearly to millions of samples and/or features.

**1.4.5. Tips on Practical Use**

* **Avoiding data copy**: For [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), [**SVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR), [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC) and [**NuSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html#sklearn.svm.NuSVR), if the data passed to certain methods is not C-ordered contiguous, and double precision, it will be copied before calling the underlying C implementation. You can check whether a given numpy array is C-contiguous by inspecting its flags attribute.

For [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) (and [**LogisticRegression**](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression)) any input passed as a numpy array will be copied and converted to the liblinear internal sparse data representation (double precision floats and int32 indices of non-zero components). If you want to fit a large-scale linear classifier without copying a dense numpy C-contiguous double precision array as input we suggest to use the [**SGDClassifier**](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html#sklearn.linear_model.SGDClassifier) class instead. The objective function can be configured to be almost the same as the [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) model.

* **Kernel cache size**: For [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), [**SVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVR.html#sklearn.svm.SVR), **nuSVC** and [**NuSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html#sklearn.svm.NuSVR), the size of the kernel cache has a strong impact on run times for larger problems. If you have enough RAM available, it is recommended to set cache\_size to a higher value than the default of 200(MB), such as 500(MB) or 1000(MB).
* **Setting C**: C is 1 by default and it’s a reasonable default choice. If you have a lot of noisy observations you should decrease it. It corresponds to regularize more the estimation.
* Support Vector Machine algorithms are not scale invariant, so **it is highly recommended to scale your data**. For example, scale each attribute on the input vector X to [0,1] or [-1,+1], or standardize it to have mean 0 and variance 1. Note that the *same* scaling must be applied to the test vector to obtain meaningful results. See section [Preprocessing data](http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing) for more details on scaling and normalization.
* Parameter nu in [**NuSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVC.html#sklearn.svm.NuSVC)/[**OneClassSVM**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html#sklearn.svm.OneClassSVM)/[**NuSVR**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.NuSVR.html#sklearn.svm.NuSVR) approximates the fraction of training errors and support vectors.
* In [**SVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC), if data for classification are unbalanced (e.g. many positive and few negative), set class\_weight='balanced' and/or try different penalty parameters C.
* The underlying [**LinearSVC**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC) 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.
* Using L1 penalization as provided by LinearSVC(loss='l2', penalty='l1', dual=False) yields a sparse solution, i.e. only a subset of feature weights is different from zero and contribute to the decision function. Increasing C yields a more complex model (more feature are selected). The C value that yields a “null” model (all weights equal to zero) can be calculated using [**l1\_min\_c**](http://scikit-learn.org/stable/modules/generated/sklearn.svm.l1_min_c.html#sklearn.svm.l1_min_c).

**1.4.6. Kernel functions**

The *kernel function* can be any of the following:

* linear: \langle x, x'\rangle.
* polynomial: (\gamma \langle x, x'\rangle + r)^d. d is specified by keyword degree, r by coef0.
* rbf: \exp(-\gamma \|x-x'\|^2). \gamma is specified by keyword gamma, must be greater than 0.
* sigmoid (\tanh(\gamma \langle x,x'\rangle + r)), where r is specified by coef0.

Different kernels are specified by keyword kernel at initialization:

>>>

**>>>** linear\_svc = svm.SVC(kernel='linear')

**>>>** linear\_svc.kernel

'linear'

**>>>** rbf\_svc = svm.SVC(kernel='rbf')

**>>>** rbf\_svc.kernel

'rbf'

**1.4.6.1. Custom Kernels**

You can define your own kernels by either giving the kernel as a python function or by precomputing the Gram matrix.

Classifiers with custom kernels behave the same way as any other classifiers, except that:

* Field support\_vectors\_ is now empty, only indices of support vectors are stored in support\_
* A reference (and not a copy) of the first argument in the fit() method is stored for future reference. If that array changes between the use of fit() and predict() you will have unexpected results.

1.4.6.1.1. Using Python functions as kernels

You can also use your own defined kernels by passing a function to the keyword kernel in the constructor.

Your kernel must take as arguments two matrices of shape (n\_samples\_1, n\_features), (n\_samples\_2, n\_features) and return a kernel matrix of shape (n\_samples\_1, n\_samples\_2).

The following code defines a linear kernel and creates a classifier instance that will use that kernel:

>>>

**>>> import** **numpy** **as** **np**

**>>> from** **sklearn** **import** svm

**>>> def** my\_kernel(X, Y):

**...**  **return** np.dot(X, Y.T)

**...**

**>>>** clf = svm.SVC(kernel=my\_kernel)

**Examples:**

* [SVM with custom kernel](http://scikit-learn.org/stable/auto_examples/svm/plot_custom_kernel.html#sphx-glr-auto-examples-svm-plot-custom-kernel-py).

1.4.6.1.2. Using the Gram matrix

Set kernel='precomputed' and pass the Gram matrix instead of X in the fit method. At the moment, the kernel values between *all* training vectors and the test vectors must be provided.

>>>

**>>> import** **numpy** **as** **np**

**>>> from** **sklearn** **import** svm

**>>>** X = np.array([[0, 0], [1, 1]])

**>>>** y = [0, 1]

**>>>** clf = svm.SVC(kernel='precomputed')

**>>>** *# linear kernel computation*

**>>>** gram = np.dot(X, X.T)

**>>>** clf.fit(gram, y)

SVC(C=1.0, cache\_size=200, class\_weight=None, coef0=0.0,

decision\_function\_shape='ovr', degree=3, gamma='auto',

kernel='precomputed', max\_iter=-1, probability=False,

random\_state=None, shrinking=True, tol=0.001, verbose=False)

**>>>** *# predict on training examples*

**>>>** clf.predict(gram)

array([0, 1])

1.4.6.1.3. Parameters of the RBF Kernel

When training an SVM with the *Radial Basis Function* (RBF) kernel, two parameters must be considered: C and gamma. The parameter C, common to all SVM kernels, trades off misclassification of training examples against simplicity of the decision surface. A low C makes the decision surface smooth, while a high C aims at classifying all training examples correctly. gamma defines how much influence a single training example has. The larger gamma is, the closer other examples must be to be affected.

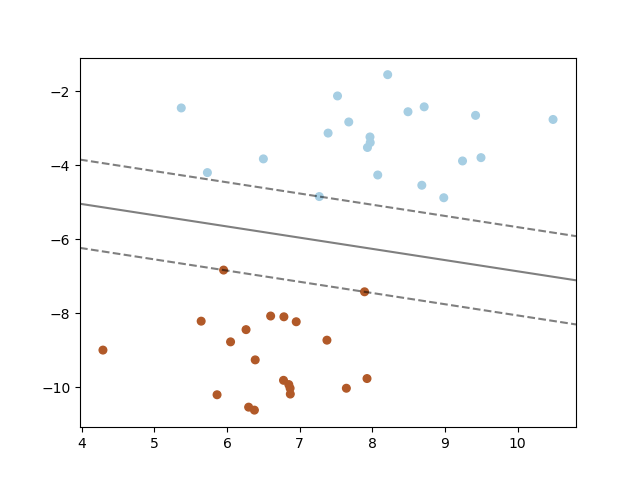
Proper choice of C and gamma is critical to the SVM’s performance. One is advised to use [**sklearn.model\_selection.GridSearchCV**](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html#sklearn.model_selection.GridSearchCV) with C and gamma spaced exponentially far apart to choose good values.

**Examples:**

* [RBF SVM parameters](http://scikit-learn.org/stable/auto_examples/svm/plot_rbf_parameters.html#sphx-glr-auto-examples-svm-plot-rbf-parameters-py)

**1.4.7. Mathematical formulation**

A support vector machine constructs a hyper-plane or set of hyper-planes in a high or infinite dimensional space, which can be used for classification, regression or other tasks. Intuitively, a good separation is achieved by the hyper-plane that has the largest distance to the nearest training data points of any class (so-called functional margin), since in general the larger the margin the lower the generalization error of the classifier.

[](http://scikit-learn.org/stable/_images/sphx_glr_plot_separating_hyperplane_0011.png)

**1.4.7.1. SVC**

Given training vectors x_i \in \mathbb{R}^p, i=1,…, n, in two classes, and a vector y \in \{1, -1\}^n, SVC solves the following primal problem:

\min_ {w, b, \zeta} \frac{1}{2} w^T w + C \sum_{i=1}^{n} \zeta_i



\textrm {subject to } & y_i (w^T \phi (x_i) + b) \geq 1 - \zeta_i,\\
& \zeta_i \geq 0, i=1, ..., n

Its dual is

\min_{\alpha} \frac{1}{2} \alpha^T Q \alpha - e^T \alpha


\textrm {subject to } & y^T \alpha = 0\\
& 0 \leq \alpha_i \leq C, i=1, ..., n

where e is the vector of all ones, C > 0 is the upper bound, Q is an n by n positive semidefinite matrix, Q_{ij} \equiv y_i y_j K(x_i, x_j), where K(x_i, x_j) = \phi (x_i)^T \phi (x_j) is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function \phi.

The decision function is:

\operatorname{sgn}(\sum_{i=1}^n y_i \alpha_i K(x_i, x) + \rho)

**Note**

While SVM models derived from [libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/) and [liblinear](http://www.csie.ntu.edu.tw/~cjlin/liblinear/) use C as regularization parameter, most other estimators use alpha. The exact equivalence between the amount of regularization of two models depends on the exact objective function optimized by the model. For example, when the estimator used is **sklearn.linear\_model.Ridge** regression, the relation between them is given as C = \frac{1}{alpha}.

This parameters can be accessed through the members dual\_coef\_ which holds the product y_i \alpha_i, support\_vectors\_which holds the support vectors, and intercept\_ which holds the independent term \rho :

**References:**

* [“Automatic Capacity Tuning of Very Large VC-dimension Classifiers”](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.7215), I. Guyon, B. Boser, V. Vapnik - Advances in neural information processing 1993.
* [“Support-vector networks”](http://link.springer.com/article/10.1007%2FBF00994018), C. Cortes, V. Vapnik - Machine Learning, 20, 273-297 (1995).

**1.4.7.2. NuSVC**

We introduce a new parameter \nu which controls the number of support vectors and training errors. The parameter \nu \in (0,
1] is an upper bound on the fraction of training errors and a lower bound of the fraction of support vectors.

It can be shown that the \nu-SVC formulation is a reparameterization of the C-SVC and therefore mathematically equivalent.

**1.4.7.3. SVR**

Given training vectors x_i \in \mathbb{R}^p, i=1,…, n, and a vector y \in \mathbb{R}^n \varepsilon-SVR solves the following primal problem:

\min_ {w, b, \zeta, \zeta^*} \frac{1}{2} w^T w + C \sum_{i=1}^{n} (\zeta_i + \zeta_i^*)



\textrm {subject to } & y_i - w^T \phi (x_i) - b \leq \varepsilon + \zeta_i,\\
                      & w^T \phi (x_i) + b - y_i \leq \varepsilon + \zeta_i^*,\\
                      & \zeta_i, \zeta_i^* \geq 0, i=1, ..., n

Its dual is

\min_{\alpha, \alpha^*} \frac{1}{2} (\alpha - \alpha^*)^T Q (\alpha - \alpha^*) + \varepsilon e^T (\alpha + \alpha^*) - y^T (\alpha - \alpha^*)


\textrm {subject to } & e^T (\alpha - \alpha^*) = 0\\
& 0 \leq \alpha_i, \alpha_i^* \leq C, i=1, ..., n

where e is the vector of all ones, C > 0 is the upper bound, Q is an n by n positive semidefinite matrix, Q_{ij} \equiv K(x_i, x_j) = \phi (x_i)^T \phi (x_j) is the kernel. Here training vectors are implicitly mapped into a higher (maybe infinite) dimensional space by the function \phi.

The decision function is:

\sum_{i=1}^n (\alpha_i - \alpha_i^*) K(x_i, x) + \rho

These parameters can be accessed through the members dual\_coef\_ which holds the difference \alpha_i - \alpha_i^*, support\_vectors\_ which holds the support vectors, and intercept\_ which holds the independent term \rho

**References:**

* [“A Tutorial on Support Vector Regression”](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.114.4288), Alex J. Smola, Bernhard Schölkopf - Statistics and Computing archive Volume 14 Issue 3, August 2004, p. 199-222.

**1.4.8. Implementation details**

Internally, we use [libsvm](http://www.csie.ntu.edu.tw/~cjlin/libsvm/) and [liblinear](http://www.csie.ntu.edu.tw/~cjlin/liblinear/) to handle all computations. These libraries are wrapped using C and Cython.