INF519 Machine Learning 2: Practice 1

Support Vector Machines (SVM)

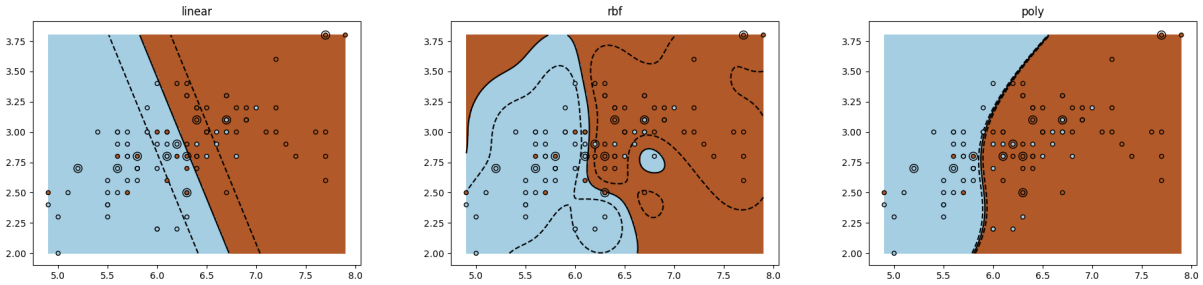
Jae Yun JUN KIM*

September 28, 2017

1 Support Vector Machines (SVMs)

Sources: Scikit-learn

1.1 Example 1: SVM classification



(a) Linear (b) RBF (c) Polynomial

Figure 1: SVM

```
print(__doc__)


import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets, svm

iris = datasets.load_iris()
X = iris.data
y = iris.target

X = X[y != 0, :2]
```

*ECE Paris Graduate School of Engineering, 37 quai de Grenelle CS71520 75 725 Paris 15, France; jae-yun.jun-kim@ece.fr

```
y = y[y != 0]

n_sample = len(X)

np.random.seed(0)
order = np.random.permutation(n_sample)
X = X[order]
y = y[order].astype(np.float)

X_train = X[:int(.9 * n_sample)]
y_train = y[:int(.9 * n_sample)]
X_test = X[int(.9 * n_sample):]
y_test = y[int(.9 * n_sample):]

# fit the model
for fig_num, kernel in enumerate(('linear', 'rbf', 'poly')):
    clf = svm.SVC(kernel=kernel, gamma=10)
    clf.fit(X_train, y_train)

    plt.figure(fig_num)
    plt.clf()
    plt.scatter(X[:, 0], X[:, 1], c=y, zorder=10, cmap=plt.cm.Paired,
                edgecolor='k', s=20)

    # Circle out the test data
    plt.scatter(X_test[:, 0], X_test[:, 1], s=80, facecolors='none',
                zorder=10, edgecolor='k')

    plt.axis('tight')
    x_min = X[:, 0].min()
    x_max = X[:, 0].max()
    y_min = X[:, 1].min()
    y_max = X[:, 1].max()

    XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
    Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(XX.shape)
    plt.pcolormesh(XX, YY, Z > 0, cmap=plt.cm.Paired)
    plt.contour(XX, YY, Z, colors=['k', 'k', 'k'],
                linestyles=['--', '-', '--'], levels=[-.5, 0, .5])

    plt.title(kernel)
plt.show()
```

1.2 Example 2: Plot different SVM classifiers in the iris dataset

Comparison of different linear SVM classifiers on a 2D projection of the iris dataset. We only consider the first 2 features of this dataset:

- Sepal length
- Sepal width

This example shows how to plot the decision surface for four SVM classifiers with different kernels. The linear models `LinearSVC()` and `SVC(kernel='linear')` yield slightly different decision boundaries. This can be a consequence of the following differences:

- LinearSVC minimizes the squared hinge loss while SVC minimizes the regular hinge loss.
- LinearSVC uses the One-vs-All (also known as One-vs-Rest) multiclass reduction while SVC uses the One-vs-One multiclass reduction.

Both linear models have linear decision boundaries (intersecting hyperplanes) while the non-linear kernel models (polynomial or Gaussian RBF) have more flexible non-linear decision boundaries with shapes that depend on the kind of kernel and its parameters.

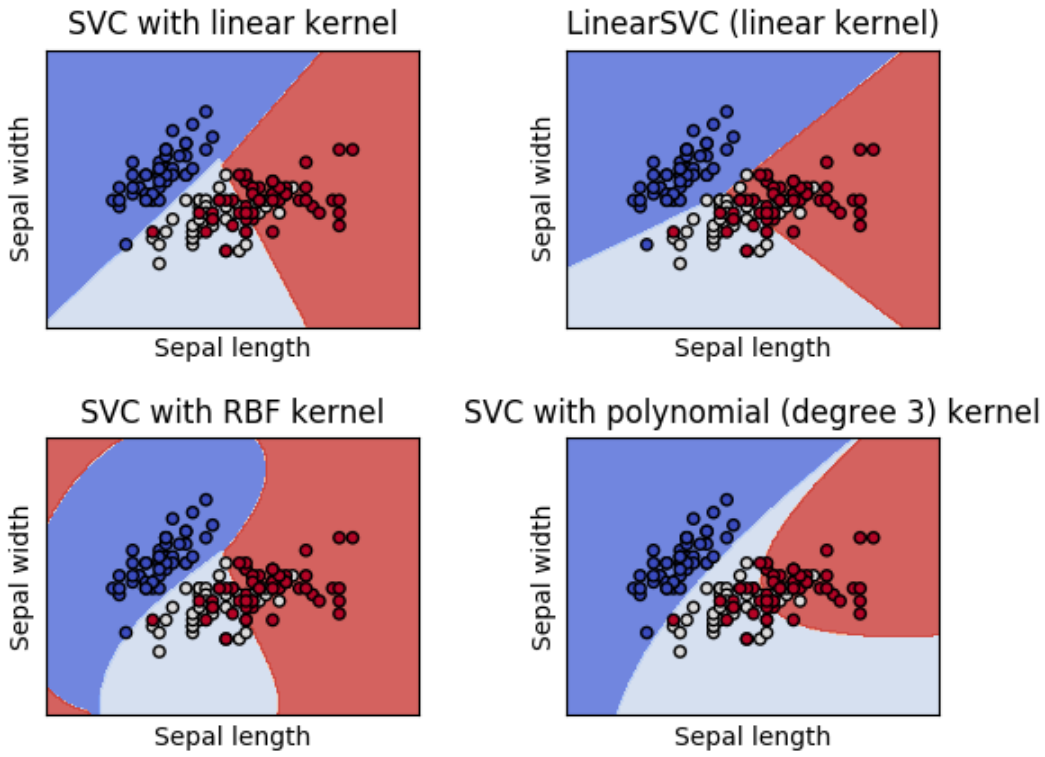


Figure 2: SVM example

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, datasets
```

```
def make_meshgrid(x, y, h=.02):
    """Create a mesh of points to plot in

    Parameters
    ----------
    x: data to base x-axis meshgrid on
    y: data to base y-axis meshgrid on
    h: stepsize for meshgrid, optional

    Returns
    -------
    xx, yy : ndarray
    """
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
    return xx, yy


def plot_contours(ax, clf, xx, yy, **params):
    """Plot the decision boundaries for a classifier.

    Parameters
    ----------
    ax: matplotlib axes object
    clf: a classifier
    xx: meshgrid ndarray
    yy: meshgrid ndarray
    params: dictionary of params to pass to contourf, optional
    """
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    out = ax.contourf(xx, yy, Z, **params)
    return out


# import some data to play with
iris = datasets.load_iris()
# Take the first two features. We could avoid this by using a two-dim dataset
X = iris.data[:, :2]
y = iris.target

# we create an instance of SVM and fit out data. We do not scale our
# data since we want to plot the support vectors
C = 1.0  # SVM regularization parameter
```

```
models = (svm.SVC(kernel='linear', C=C),
          svm.LinearSVC(C=C),
          svm.SVC(kernel='rbf', gamma=0.7, C=C),
          svm.SVC(kernel='poly', degree=3, C=C))
models = (clf.fit(X, y) for clf in models)

# title for the plots
titles = ('SVC with linear kernel',
          'LinearSVC (linear kernel)',
          'SVC with RBF kernel',
          'SVC with polynomial (degree 3) kernel')

# Set-up 2x2 grid for plotting.
fig, sub = plt.subplots(2, 2)
plt.subplots_adjust(wspace=0.4, hspace=0.4)

X0, X1 = X[:, 0], X[:, 1]
xx, yy = make_meshgrid(X0, X1)

for clf, title, ax in zip(models, titles, sub.flatten()):
    plot_contours(ax, clf, xx, yy,
                  cmap=plt.cm.coolwarm, alpha=0.8)
    ax.scatter(X0, X1, c=y, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xlabel('Sepal length')
    ax.set_ylabel('Sepal width')
    ax.set_xticks(())
    ax.set_yticks(())
    ax.set_title(title)

plt.show()
```

1.3 Example 3: Nonlinear classification

Perform binary classification using non-linear SVC with RBF kernel. The target to predict is a XOR of the inputs.
The color map illustrates the decision function learned by the SVC.

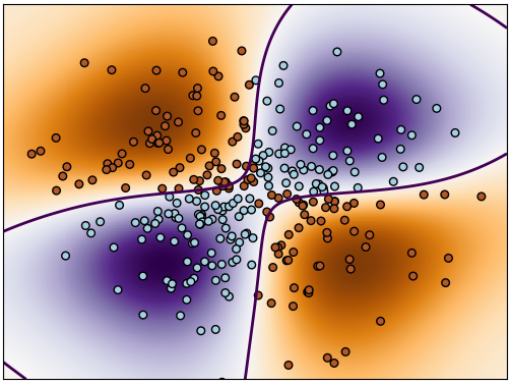


Figure 3: Nonlinear SVM example

```
print(__doc__)

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

xx, yy = np.meshgrid(np.linspace(-3, 3, 500),
                     np.linspace(-3, 3, 500))
np.random.seed(0)
X = np.random.randn(300, 2)
Y = np.logical_xor(X[:, 0] > 0, X[:, 1] > 0)

# fit the model
clf = svm.NuSVC()
clf.fit(X, Y)

# plot the decision function for each datapoint on the grid
Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

plt.imshow(Z, interpolation='nearest',
           extent=(xx.min(), xx.max(), yy.min(), yy.max()), aspect='auto',
           origin='lower', cmap=plt.cm.PuOr_r)
contours = plt.contour(xx, yy, Z, levels=[0], linewidths=2,
                       linetypes='--')
plt.scatter(X[:, 0], X[:, 1], s=30, c=Y, cmap=plt.cm.Paired,
            edgecolors='k')
plt.xticks(())
plt.yticks(())
plt.axis([-3, 3, -3, 3])
```

```
plt.show()
```

1.4 sklearn.svm.SVC

```
class sklearn.svm.SVC(C=1.0, kernel='rbf', degree=3, gamma='auto',
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', random_state=None)
```

C-Support Vector Classification.

The implementation is based on `libsvm`. The fit time complexity is more than quadratic with the number of samples which makes it hard to scale to dataset with more than a couple of 10000 samples.

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

1.4.1 Parameters

C: float, optional (default:1.0)

Penalty parameter **C** of the error term.

kernel: string, optional (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, optional (default:3)

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

gamma: float, optional (default:'auto')

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma is 'auto' then `1/n_features` will be used instead.

coef0: float, optional (default:0.0)

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

probability: boolean, optional (default:False)

Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.

shrinking: boolean, optional (default:True)

Whether to use the shrinking heuristic.

tol: float, optional (default:1e-3)

Tolerance for stopping criterion.

cache_size: float, optional

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

class_weight: dict, 'balanced', optional

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, optional (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, optional (default:-1)

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

decision_function_shape: float, optional (default:0.0)

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

random_state: int, RandomState instance or None, optional (default:False)

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`.

1.4.2 Attributes

support_: array-like, shape=[n_SV]

Indices of support vectors

support_vectors_: array-like, shape=[n_SV,n_features]

Support vectors.

n_support_: array-like, dtype=int32, shape=[n_class]

Number of support vectors for each class.

dual_coef_: array, shape=[n_class-1, n_SV]

Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_: array, shape = [n_class-1, 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 read-only property derived from `dual_coef_` and `support_vectors_`.

intercept_: array, shape=[n_class * (n_class-1)/2]

Constants in decision function.

1.4.3 Methods

```
__init__(C=1.0, kernel='rbf', degree=3, gamma='auto', 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', random_state=None)
```

```
decision_function(X)
```

Distance of the samples X to the separating hyperplane.

Parameters: **X**: array-like, shape (n_samples, n_features)

Returns: **X**: array-like, shape (n_samples, n_classes * (n_classes-1)/2)
Returns the decision function of the sample for each class in the model.
If `decision_function_shape='ovr'`, the shape is (`n_samples`, `n_classes`).

```
fit(X, y, sample_weight=None)
```

Fit the SVM model according to the given training data.

Parameters:

X: {array-like, shape} (n_samples, n_features)

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features. For `kernel=''precomputed"`, the expected shape of X is (`n_samples`, `n_samples`).

y: array-like, shape (n_samples)

Target values (class labels in classification, real numbers in regression).

sample_weight: array-like, shape (n_samples)

Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns: **self**: object

Returns `self`.

```
get_params(deep=True)
```

Get parameters for this estimator.
Params:

deep: boolean, optional

If True, will return the parameters for this estimator and contained `subobjects` that are estimators.

Returns:

params: mapping of string to any

Parameter names mapped to their values.

```
predict(X)
```

Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.
Parameters:

X: array-like, sparse matrix, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

y_pred: array, shape (n_samples,)

Class labels for samples in X.

```
predict_log_proba(X)
```

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute probability set to True.
Parameters:

X: array-like, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

T: array-like, shape (n_samples,n_classes)

Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

`predict_proba(X)`

Compute probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute probability set to True.
Parameters:

X: array-like, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

T: array-like, shape (n_samples,n_classes)

Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

`score(X,y,sample_weight=None)`

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.
Parameters:

X: array-like, shape = (n_samples, n_features)

Test samples

y: array-like, shape = (n_samples) or , (n_samples,n_features)

True labels for X.

sample_weight: array-like, shape = [n_samples], optional

Sample weights

Returns:

score: float

Mean accuracy of `self.predict(X)` w.r.t. y.

`set_params(**params)`

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form $< component > \_\_ < parameter >$ so that it is possible to update each component of a nested object.
Returns:

self: object

Returns self.

1.5 sklearn.svm.LinearSVC

```
class sklearn.svm.LinearSVC(penalty='l2', loss='squared_hinge', dual=True,
tol=0.0001, C=1.0, multi_class='ovr', fit_intercept=True, intercept_scaling=1,
class_weight=None, verbose=0, random_state=None, max_iter=1000)
```

Linear Support Vector Classification.

Similar to SVC with parameter `kernel='linear'`, but implemented in terms of `liblinear` rather than `libsvm`, so it has more flexibility in the choice of penalties and loss functions and should scale better to large numbers of samples.

This class supports both dense and sparse input and the multiclass support is handled according to a one-vs-the-rest scheme.

1.5.1 Parameters

penalty: string, 'l1' or 'l2' (default='l2')

Specifies the norm used in the penalization. The 'l2' penalty is the standard used in SVC. The 'l1' leads to `coef_` vectors that are sparse.

loss: string, 'hinge' or 'squared_hinge' (default='squared_hinge')

Specifies the loss function. 'hinge' is the standard SVM loss (used e.g. by the SVC class) while 'squared_hinge' is the square of the hinge loss.

dual: bool, (default=True)

Select the algorithm to either solve the dual or primal optimization problem. Prefer `dual=False` when n_samples ¿ n_features.

tol: float, optional (default=1e-3)

Tolerance for stopping criterion.

C: float, optional (default=1.0)

Penalty parameter C of the error term.

multi_class: string, 'ovr' or 'crammer_singer' (default='ovr')

Determines the multi-class strategy if y contains more than two classes. "ovr" trains n_classes one-vs-rest classifiers, while "crammer_singer" optimizes a joint objective over all classes. While crammer_singer is interesting from a theoretical perspective as it is consistent, it is seldom used in practice as it rarely leads to better accuracy and is more expensive to compute. If "crammer_singer" is chosen, the options loss, penalty and dual will be ignored.

probability: boolean, optional (default:False)

Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.

fit_intercept: boolean, optional (default=True)

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

intercept_scaling: float, optional (default=1)

When `self.fit_intercept` is True, instance vector 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.

class_weight: dict, 'balanced', optional

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, optional (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.

random_state: int, RandomState instance or None, optional (default:False)

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`.

max_iter: int, (default:1000)

The maximum number of iterations to be run.

1.5.2 Attributes

coef_: array, shape=[n_features] if $n\_classes == 2$, else [n_classes, 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 `raw_coef_` that follows the internal memory layout of liblinear.

intercept_: array, shape=[1], if n_classes==2, else [n_classes].

Constants in decision function.

1.5.3 Methods

```
__init__(penalty='l2', loss='squared_hinge', dual=True, tol=0.0001, C=1.0,
multi_class='ovr', fit_intercept=True, intercept_scaling=1,
class_weight=None, verbose=0, random_state=None, max_iter=1000)
```

```
decision_function(X)
```

Distance of the samples X to the separating hyperplane.

Parameters: **X**: array-like, shape (n_samples, n_features)

Returns:

X: array, shape=(n_samples,) if n_classes==2, else (n_samples, n_classes)

Confidence scores per (sample, class) combination. In the binary case, confidence score for self.classes_[1] where ¿ 0 means this class would be predicted.

```
densify()
```

Convert coefficient matrix to dense array format.

Converts the `coef_` member (back) to a `numpy.ndarray`. This is the default format of `coef_` and is required for fitting, so calling this method is only required on models that have previously been sparsified; otherwise, it is a no-op.

Returns:

self: estimator

```
fit(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 vectors, where `n_samples` is the number of samples and `n_features` is the number of features.

y: array-like, shape=[n_samples]

Target values 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

Returns `self`.

`get_params(deep=True)`

Get parameters for this estimator.
Params:

deep: boolean, optional

If True, will return the parameters for this estimator and contained `subobjects` that are estimators.

Returns:

params: mapping of string to any

Parameter names mapped to their values.

`predict(X)`

Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.
Parameters:

X: array-like, sparse matrix, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

C: array, shape=[n_samples,]

Predicted class label per sample.

`score(X,y,sample_weight=None)`

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.
Parameters:

X: array-like, shape = (n_samples, n_features)

Test samples

y: array-like, shape = (n_samples) or , (n_samples,n_features)

True labels for X.

sample_weight: array-like, shape = [n_samples], optional

Sample weights

Returns:

score: float

Mean accuracy of `self.predict(X)` w.r.t. y.

```
set_params(**params)
```

Set the parameters of this estimator.

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

Returns:

self: object

Returns self.

```
sparsify()
```

Convert coefficient matrix to sparse format.

Converts the `coef_` member to a scipy.sparse matrix, which for L1-regularized models can be much more memory- and storage-efficient than the usual `numpy.ndarray` representation.

The `intercept_` member is not converted.

Returns:

self: object

Returns self.

1.6 sklearn.svm.NuSVC

```
class sklearn.svm.NuSVC(nu=0.5, kernel='rbf', degree=3, gamma='auto',
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', random_state=None)
```

Nu-Support Vector Classification.

Similar to SVC but uses a parameter to control the number of support vectors.

The implementation is based on `libsvm`.

1.6.1 Parameters

nu: float, optional (default:0.5)

An upper bound on the fraction of training errors and a lower bound of the fraction of support vectors. Should be in the interval (0, 1].

kernel: string, optional (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, optional (default:3)

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

gamma: float, optional (default:'auto')

Kernel coefficient for 'rbf', 'poly' and 'sigmoid'. If gamma is 'auto' then `1/n_features` will be used instead.

coef0: float, optional (default:0.0)

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

probability: boolean, optional (default:False)

Whether to enable probability estimates. This must be enabled prior to calling fit, and will slow down that method.

shrinking: boolean, optional (default:True)

Whether to use the shrinking heuristic.

tol: float, optional (default:1e-3)

Tolerance for stopping criterion.

cache_size: float, optional

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

class_weight: dict, 'balanced', optional

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, optional (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, optional (default:-1)

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

decision_function_shape: float, optional (default:0.0)

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

random_state: int, RandomState instance or None, optional (default:False)

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`.

1.6.2 Attributes

support_: array-like, shape=[n_SV]

Indices of support vectors

support_vectors_: array-like, shape=[n_SV,n_features]

Support vectors.

n_support_: array-like, dtype=int32, shape=[n_class]

Number of support vectors for each class.

dual_coef_: array, shape=[n_class-1, n_SV]

Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_: array, shape = [n_class-1, 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 read-only property derived from `dual_coef_` and `support_vectors_`.

intercept_: array, shape=[n_class * (n_class-1)/2]

Constants in decision function.

1.6.3 Methods

```
__init__(nu=0.5, kernel='rbf', degree=3, gamma='auto', 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', random_state=None)
```

```
decision_function(X)
```

Distance of the samples X to the separating hyperplane.

Parameters: **X**: array-like, shape (n_samples, n_features)

Returns: **X**: array-like, shape (n_samples, n_classes * (n_classes-1)/2)
Returns the decision function of the sample for each class in the model.
If `decision_function_shape='ovr'`, the shape is (`n_samples`, `n_classes`).

```
fit(X, y, sample_weight=None)
```

Fit the SVM model according to the given training data.

Parameters:

X: {array-like, shape} (n_samples, n_features)

Training vectors, where `n_samples` is the number of samples and `n_features` is the number of features. For `kernel='precomputed'`, the expected shape of X is (`n_samples`, `n_samples`).

y: array-like, shape (n_samples)

Target values (class labels in classification, real numbers in regression).

sample_weight: array-like, shape (n_samples)

Per-sample weights. Rescale C per sample. Higher weights force the classifier to put more emphasis on these points.

Returns: **self**: object

Returns `self`.

```
get_params(deep=True)
```

Get parameters for this estimator.
Params:

deep: boolean, optional

If True, will return the parameters for this estimator and contained `subobjects` that are estimators.

Returns:

params: mapping of string to any

Parameter names mapped to their values.

```
predict(X)
```

Perform classification on samples in X.

For an one-class model, +1 or -1 is returned.
Parameters:

X: array-like, sparse matrix, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

y_pred: array, shape (n_samples,)

Class labels for samples in X.

```
predict_log_proba(X)
```

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute probability set to True.

Parameters:

X: array-like, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

T: array-like, shape (n_samples,n_classes)

Returns the log-probabilities of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
predict\_proba(X)
```

Compute log probabilities of possible outcomes for samples in X.

The model need to have probability information computed at training time: fit with attribute probability set to True.

Parameters:

X: array-like, shape (n_samples, n_features)

For `kernel=‘‘precomputed"`, the expected shape of X is [n_samples_test, n_samples_train]

Returns:

T: array-like, shape (n_samples,n_classes)

Returns the probability of the sample for each class in the model. The columns correspond to the classes in sorted order, as they appear in the attribute `classes_`.

```
score(X,y,sample_weight=None)
```

Returns the mean accuracy on the given test data and labels.

In multi-label classification, this is the subset accuracy which is a harsh metric since you require for each sample that each label set be correctly predicted.

Parameters:

X: array-like, shape = (n_samples, n_features)

Test samples

y: array-like, shape = (n_samples) or , (n_samples,n_features)

True labels for X.

sample_weight: array-like, shape = [n_samples], optional

Sample weights

Returns:

score: float

Mean accuracy of `self.predict(X)` w.r.t. y.

```
set_params(**params)
```

Set the parameters of this estimator.

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

Returns:

self: object

Returns self.

1.7 One-vs.-rest and One-vs.-one

One-vs.-rest

The one-vs.-rest strategy involves training a single classifier per class, with the samples of that class as positive samples and all other samples as negatives. This strategy requires the base classifiers to produce a real-valued confidence score for its decision, rather than just a class label; discrete class labels alone can lead to ambiguities, where multiple classes are predicted for a single sample

One-vs.-one

In the one-vs.-one (OvO) reduction, one trains K (K 1) / 2 binary classifiers for a K-way multiclass problem; each receives the samples of a pair of classes from the original training set, and must learn to distinguish these two classes. At prediction time, a voting scheme is applied: all K (K 1) / 2 classifiers are applied to an unseen sample and the class that got the highest number of "+1" predictions gets predicted by the combined classifier.