GenSVM Documentation

Release # -*- coding: utf-8 -*-

Gertjan van den Burg

Oct 27, 2020

Contents

This is the Python package for the GenSVM multiclass classifier by Gerrit J.J. van den Burg and Patrick J.F. Groenen.

Useful links:

- PyGenSVM on GitHub
- PyGenSVM on PyPI
- Package documentation
- Journal paper: GenSVM: A Generalized Multiclass Support Vector Machine JMLR, 17(225):142, 2016.
- There is also an R package
- Or you can directly use the C library

CHAPTER 1

Installation

Before GenSVM can be installed, a working NumPy installation is required. so GenSVM can be installed using the following command:

```
$ pip install numpy && pip install gensvm
```

If you encounter any errors, please open an issue on GitHub. Don't hesitate, you're helping to make this project better!

CHAPTER 2

Citing

If you use this package in your research please cite the paper, for instance using the following BibTeX entry:

```
@article{JMLR:v17:14-526,
        author  = {{van den Burg}, G. J. J. and Groenen, P. J. F.},
        title   = {{GenSVM}: A Generalized Multiclass Support Vector Machine},
        journal = {Journal of Machine Learning Research},
        year    = {2016},
        volume  = {17},
        number  = {225},
        pages   = {1-42},
        url     = {http://jmlr.org/papers/v17/14-526.html}
}
```

CHAPTER 3

Usage

The package contains two classes to fit the GenSVM model: GenSVM and GenSVMGridSearchCV. These classes respectively fit a single GenSVM model or fit a series of models for a parameter grid search. The interface to these classes is the same as that of classifiers in Scikit-Learn so users familiar with Scikit-Learn should have no trouble using this package. Below we will show some examples of using the GenSVM classifier and the GenSVMGridSearchCV class in practice.

In the examples we assume that we have loaded the iris dataset from Scikit-Learn as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import MaxAbsScaler
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> scaler = MaxAbsScaler().fit(X_train)
>>> X_train, X_test = scaler.transform(X_train), scaler.transform(X_test)
```

Note that we scale the data using the MaxAbsScaler function. This scales the columns of the data matrix to `[-1, 1]` without breaking sparsity. Scaling the dataset can have a significant effect on the computation time of GenSVM and is generally recommended for SVMs.

3.1 Example 1: Fitting a single GenSVM model

Let's start by fitting the most basic GenSVM model on the training data:

```
>>> from gensvm import GenSVM
>>> clf = GenSVM()
>>> clf.fit(X_train, y_train)
GenSVM(coef=0.0, degree=2.0, epsilon=1e-06, gamma='auto', kappa=0.0,
kernel='linear', kernel_eigen_cutoff=1e-08, lmd=1e-05,
max_iter=100000000.0, p=1.0, random_state=None, verbose=0,
weights='unit')
```

With the model fitted, we can predict the test dataset:

```
>>> y_pred = clf.predict(X_test)
```

Next, we can compute a score for the predictions. The GenSVM class has a `score` method which computes the accuracy_score for the predictions. In the GenSVM paper, the adjusted Rand index is often used to compare performance. We illustrate both options below (your results may be different depending on the exact train/test split):

```
>>> clf.score(X_test, y_test)
1.0
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(clf.predict(X_test), y_test)
1.0
```

We can try this again by changing the model parameters, for instance we can turn on verbosity and use the Euclidean norm in the GenSVM model by setting `p = 2`:

```
>>> clf2 = GenSVM(verbose=True, p=2)
>>> clf2.fit(X_train, y_train)
Starting main loop.
Dataset:
    n = 112
    m = 4
    K = 3
Parameters:
    kappa = 0.000000
    p = 2.000000
    lambda = 0.0000100000000000
    epsilon = 1e-06

iter = 0, L = 3.4499531579689533, Lbar = 7.3369415851139745, reldiff = 1.
↪1266786095824437
...
Optimization finished, iter = 4046, loss = 0.0230726364692517, rel. diff. = 0.
↪0000009998645783
Number of support vectors: 9
GenSVM(coef=0.0, degree=2.0, epsilon=1e-06, gamma='auto', kappa=0.0,
    kernel='linear', kernel_eigen_cutoff=1e-08, lmd=1e-05,
    max_iter=100000000.0, p=2, random_state=None, verbose=True,
    weights='unit')
```

For other parameters that can be tuned in the GenSVM model, see GenSVM.

3.2 Example 2: Fitting a GenSVM model with a "warm start"

One of the key features of the GenSVM classifier is that training can be accelerated by using so-called "warm-starts". This way the optimization can be started in a location that is closer to the final solution than a random starting position would be. To support this, the `fit` method of the GenSVM class has an optional `seed_V` parameter. We'll illustrate how this can be used below.

We start with relatively large value for the `epsilon` parameter in the model. This is the stopping parameter that determines how long the optimization continues (and therefore how exact the fit is).

```
>>> clf1 = GenSVM(epsilon=1e-3)
>>> clf1.fit(X_train, y_train)
...
```

(continues on next page)

(continued from previous page)

```
>>> clf1.n_iter_
163
```

The `n_iter_` attribute tells us how many iterations the model did. Now, we can use the solution of this model to start the training for the next model:

```
>>> clf2 = GenSVM(epsilon=1e-8)
>>> clf2.fit(X_train, y_train, seed_V=clf1.combined_coef_)
...
>>> clf2.n_iter_
3196
```

Compare this to a model with the same stopping parameter, but without the warm start:

```
>>> clf2.fit(X_train, y_train)
...
>>> clf2.n_iter_
3699
```

So we saved about 500 iterations! This effect will be especially significant with large datasets and when you try out many parameter configurations. Therefore this technique is built into the GenSVMGridSearchCV class that can be used to do a grid search of parameters.

3.3 Example 3: Running a GenSVM grid search

Often when we're fitting a machine learning model such as GenSVM, we have to try several parameter configurations to figure out which one performs best on our given dataset. This is usually combined with cross validation to avoid overfitting. To do this efficiently and to make use of warm starts, the GenSVMGridSearchCV class is available. This class works in the same way as the GridSearchCV class of Scikit-Learn, but uses the GenSVM C library for speed.

To do a grid search, we first have to define the parameters that we want to vary and what values we want to try:

```
>>> from gensvm import GenSVMGridSearchCV
>>> param_grid = {'p': [1.0, 2.0], 'lmd': [1e-8, 1e-6, 1e-4, 1e-2, 1.0], 'kappa': [-0.
↪9, 0.0] }
```

For the values that are not varied in the parameter grid, the default values will be used. This means that if you want to change a specific value (such as `epsilon` for instance), you can add this to the parameter grid as a parameter with a single value to try (e.g. `'epsilon': [1e-8]`).

Running the grid search is now straightforward:

```
>>> gg = GenSVMGridSearchCV(param_grid)
>>> gg.fit(X_train, y_train)
GenSVMGridSearchCV(cv=None, iid=True,
      param_grid={'p': [1.0, 2.0], 'lmd': [1e-06, 0.0001, 0.01, 1.0], 'kappa': [-0.9,␣
↪0.0]},
      refit=True, return_train_score=True, scoring=None, verbose=0)
```

Note that if we have set `refit=True` (the default), then we can use the GenSVMGridSearchCV instance to predict or score using the best estimator found in the grid search:

```
>>> y_pred = gg.predict(X_test)
>>> gg.score(X_test, y_test)
1.0
```

A nice feature borrowed from Scikit-Learn is that the results from the grid search can be represented as a `pandas` DataFrame:

```
>>> from pandas import DataFrame
>>> df = DataFrame(gg.cv_results_)
```

This can make it easier to explore the results of the grid search.

CHAPTER 4

Known Limitations

The following are known limitations that are on the roadmap for a future release of the package. If you need any of these features, please vote on them on the linked GitHub issues (this can make us add them sooner!).

1. Support for sparse matrices. NumPy supports sparse matrices, as does the GenSVM C library. Getting them to work together requires some additional effort. In the meantime, if you really want to use sparse data with GenSVM (this can lead to significant speedups!), check out the GenSVM C library.
2. Specification of class misclassification weights. Currently, incorrectly classification an object from class A to class C is as bad as incorrectly classifying an object from class B to class C. Depending on the application, this may not be the desired effect. Adding class misclassification weights can solve this issue.

CHAPTER 5

Questions and Issues

If you have any questions or encounter any issues with using this package, please ask them on GitHub.

CHAPTER 6

License

This package is licensed under the GNU General Public License version 3.

Copyright (c) G.J.J. van den Burg, excluding the sections of the code that are explicitly marked to come from Scikit-Learn.

6.1 GenSVM Python Package

This is the Python package for the GenSVM multiclass classifier by Gerrit J.J. van den Burg and Patrick J.F. Groenen.

Useful links:

- PyGenSVM on GitHub
- PyGenSVM on PyPI
- Package documentation
- Journal paper: GenSVM: A Generalized Multiclass Support Vector Machine JMLR, 17(225):142, 2016.
- There is also an R package
- Or you can directly use the C library

6.1.1 Installation

Before GenSVM can be installed, a working NumPy installation is required. so GenSVM can be installed using the following command:

```
$ pip install numpy && pip install gensvm
```

If you encounter any errors, please open an issue on GitHub. Don't hesitate, you're helping to make this project better!

6.1.2 Citing

If you use this package in your research please cite the paper, for instance using the following BibTeX entry:

```
@article{JMLR:v17:14-526,
        author  = {{van den Burg}, G. J. J. and Groenen, P. J. F.},
        title   = {{GenSVM}: A Generalized Multiclass Support Vector Machine},
        journal = {Journal of Machine Learning Research},
        year    = {2016},
        volume  = {17},
        number  = {225},
        pages   = {1-42},
        url     = {http://jmlr.org/papers/v17/14-526.html}
}
```

6.1.3 Usage

The package contains two classes to fit the GenSVM model: GenSVM and GenSVMGridSearchCV. These classes respectively fit a single GenSVM model or fit a series of models for a parameter grid search. The interface to these classes is the same as that of classifiers in Scikit-Learn so users familiar with Scikit-Learn should have no trouble using this package. Below we will show some examples of using the GenSVM classifier and the GenSVMGridSearchCV class in practice.

In the examples we assume that we have loaded the iris dataset from Scikit-Learn as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.preprocessing import MaxAbsScaler
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X, y)
>>> scaler = MaxAbsScaler().fit(X_train)
>>> X_train, X_test = scaler.transform(X_train), scaler.transform(X_test)
```

Note that we scale the data using the MaxAbsScaler function. This scales the columns of the data matrix to `[-1, 1]` without breaking sparsity. Scaling the dataset can have a significant effect on the computation time of GenSVM and is generally recommended for SVMs.

Example 1: Fitting a single GenSVM model

Let's start by fitting the most basic GenSVM model on the training data:

```
>>> from gensvm import GenSVM
>>> clf = GenSVM()
>>> clf.fit(X_train, y_train)
GenSVM(coef=0.0, degree=2.0, epsilon=1e-06, gamma='auto', kappa=0.0,
kernel='linear', kernel_eigen_cutoff=1e-08, lmd=1e-05,
max_iter=100000000.0, p=1.0, random_state=None, verbose=0,
weights='unit')
```

With the model fitted, we can predict the test dataset:

```
>>> y_pred = clf.predict(X_test)
```

Next, we can compute a score for the predictions. The GenSVM class has a `score` method which computes the accuracy_score for the predictions. In the GenSVM paper, the adjusted Rand index is often used to compare performance. We illustrate both options below (your results may be different depending on the exact train/test split):

```
>>> clf.score(X_test, y_test)
1.0
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(clf.predict(X_test), y_test)
1.0
```

We can try this again by changing the model parameters, for instance we can turn on verbosity and use the Euclidean norm in the GenSVM model by setting `p = 2`:

```
>>> clf2 = GenSVM(verbose=True, p=2)
>>> clf2.fit(X_train, y_train)
Starting main loop.
Dataset:
    n = 112
    m = 4
    K = 3
Parameters:
    kappa = 0.000000
    p = 2.000000
    lambda = 0.0000100000000000
    epsilon = 1e-06

iter = 0, L = 3.4499531579689533, Lbar = 7.3369415851139745, reldiff = 1.
↪1266786095824437
...
Optimization finished, iter = 4046, loss = 0.0230726364692517, rel. diff. = 0.
↪0000009998645783
Number of support vectors: 9
GenSVM(coef=0.0, degree=2.0, epsilon=1e-06, gamma='auto', kappa=0.0,
    kernel='linear', kernel_eigen_cutoff=1e-08, lmd=1e-05,
    max_iter=100000000.0, p=2, random_state=None, verbose=True,
    weights='unit')
```

For other parameters that can be tuned in the GenSVM model, see GenSVM.

Example 2: Fitting a GenSVM model with a "warm start"

One of the key features of the GenSVM classifier is that training can be accelerated by using so-called "warm-starts". This way the optimization can be started in a location that is closer to the final solution than a random starting position would be. To support this, the `fit` method of the GenSVM class has an optional `seed_V` parameter. We'll illustrate how this can be used below.

We start with relatively large value for the `epsilon` parameter in the model. This is the stopping parameter that determines how long the optimization continues (and therefore how exact the fit is).

```
>>> clf1 = GenSVM(epsilon=1e-3)
>>> clf1.fit(X_train, y_train)
...
>>> clf1.n_iter_
163
```

The `n_iter_` attribute tells us how many iterations the model did. Now, we can use the solution of this model to start the training for the next model:

```
>>> clf2 = GenSVM(epsilon=1e-8)
>>> clf2.fit(X_train, y_train, seed_V=clf1.combined_coef_)
```

(continues on next page)

(continued from previous page)

```
...
>>> clf2.n_iter_
3196
```

Compare this to a model with the same stopping parameter, but without the warm start:

```
>>> clf2.fit(X_train, y_train)
...
>>> clf2.n_iter_
3699
```

So we saved about 500 iterations! This effect will be especially significant with large datasets and when you try out many parameter configurations. Therefore this technique is built into the GenSVMGridSearchCV class that can be used to do a grid search of parameters.

Example 3: Running a GenSVM grid search

Often when we're fitting a machine learning model such as GenSVM, we have to try several parameter configurations to figure out which one performs best on our given dataset. This is usually combined with cross validation to avoid overfitting. To do this efficiently and to make use of warm starts, the GenSVMGridSearchCV class is available. This class works in the same way as the GridSearchCV class of Scikit-Learn, but uses the GenSVM C library for speed.

To do a grid search, we first have to define the parameters that we want to vary and what values we want to try:

```
>>> from gensvm import GenSVMGridSearchCV
>>> param_grid = {'p': [1.0, 2.0], 'lmd': [1e-8, 1e-6, 1e-4, 1e-2, 1.0], 'kappa': [-0.
↪9, 0.0] }
```

For the values that are not varied in the parameter grid, the default values will be used. This means that if you want to change a specific value (such as `epsilon` for instance), you can add this to the parameter grid as a parameter with a single value to try (e.g. `'epsilon': [1e-8]`).

Running the grid search is now straightforward:

```
>>> gg = GenSVMGridSearchCV(param_grid)
>>> gg.fit(X_train, y_train)
GenSVMGridSearchCV(cv=None, iid=True,
      param_grid={'p': [1.0, 2.0], 'lmd': [1e-06, 0.0001, 0.01, 1.0], 'kappa': [-0.9,␣
↪0.0]},
      refit=True, return_train_score=True, scoring=None, verbose=0)
```

Note that if we have set `refit=True` (the default), then we can use the GenSVMGridSearchCV instance to predict or score using the best estimator found in the grid search:

```
>>> y_pred = gg.predict(X_test)
>>> gg.score(X_test, y_test)
1.0
```

A nice feature borrowed from Scikit-Learn is that the results from the grid search can be represented as a `pandas` DataFrame:

```
>>> from pandas import DataFrame
>>> df = DataFrame(gg.cv_results_)
```

This can make it easier to explore the results of the grid search.

6.1.4 Known Limitations

The following are known limitations that are on the roadmap for a future release of the package. If you need any of these features, please vote on them on the linked GitHub issues (this can make us add them sooner!).

1. Support for sparse matrices. NumPy supports sparse matrices, as does the GenSVM C library. Getting them to work together requires some additional effort. In the meantime, if you really want to use sparse data with GenSVM (this can lead to significant speedups!), check out the GenSVM C library.
2. Specification of class misclassification weights. Currently, incorrectly classification an object from class A to class C is as bad as incorrectly classifying an object from class B to class C. Depending on the application, this may not be the desired effect. Adding class misclassification weights can solve this issue.

6.1.5 Questions and Issues

If you have any questions or encounter any issues with using this package, please ask them on GitHub.

6.1.6 License

This package is licensed under the GNU General Public License version 3.

Copyright (c) G.J.J. van den Burg, excluding the sections of the code that are explicitly marked to come from Scikit-Learn.

6.2 GenSVM

class gensvm.core.**GenSVM**(*p=1.0*, *lmd=1e-05*, *kappa=0.0*, *epsilon=1e-06*, *weights='unit'*, *kernel='linear'*, *gamma='auto'*, *coef=1.0*, *degree=2.0*, *kernel_eigen_cutoff=1e-08*, *verbose=0*, *random_state=None*, *max_iter=100000000.0*)

Generalized Multiclass Support Vector Machine Classification.

This class implements the basic GenSVM classifier. GenSVM is a generalized multiclass SVM which is flexible in the weighting of misclassification errors. It is this flexibility that makes it perform well on diverse datasets.

The `fit()` and `predict()` methods of this class use the GenSVM C library for the actual computations.

Parameters

- **p** (*float, optional (default=1.0)*) – Parameter for the L_p norm of the loss function (1.0 <= p <= 2.0)
- **lmd** (*float, optional (default=1e-5)*) – Parameter for the regularization term of the loss function (lmd > 0)
- **kappa** (*float, optional (default=0.0)*) – Parameter for the hinge function in the loss function (kappa > -1.0)
- **weights** (*string, optional (default='unit')*) – Type of sample weights to use. Options are 'unit' for unit weights and 'group' for group size correction weights (equation 4 in the paper).

  It is also possible to provide an explicit vector of sample weights through the `fit()` method. If so, it will override the setting provided here.
- **kernel** (*string, optional (default='linear')*) – Specify the kernel type to use in the classifier. It must be one of 'linear', 'poly', 'rbf', or 'sigmoid'.

- **gamma** (*float, optional (default='auto')*) – Kernel parameter for the rbf, poly, and sigmoid kernel. If gamma is 'auto' then 1/n_features will be used. See Kernels in GenSVM for the exact implementation of the kernels.
- **coef** (*float, optional (default=1.0)*) – Kernel parameter for the poly and sigmoid kernel. See Kernels in GenSVM for the exact implementation of the kernels.
- **degree** (*float, optional (default=2.0)*) – Kernel parameter for the poly kernel. See Kernels in GenSVM for the exact implementation of the kernels.
- **kernel_eigen_cutoff** (*float, optional (default=1e-8)*) – Cutoff point for the reduced eigendecomposition used with nonlinear GenSVM. Eigenvectors for which the ratio between their corresponding eigenvalue and the largest eigenvalue is smaller than the cutoff will be dropped.
- **verbose** (*int, (default=0)*) – Enable verbose output
- **random_state** (*None, int, instance of RandomState*) – The seed for the random number generation used for initialization where necessary. See the documentation of `sklearn.utils.check_random_state` for more info.
- **max_iter** (*int, (default=1e8)*) – The maximum number of iterations to be run.

coef_
array, shape = [n_features, n_classes-1] – Weights assigned to the features (coefficients in the primal problem)

intercept_
array, shape = [n_classes-1] – Constants in the decision function

combined_coef_
array, shape = [n_features+1, n_classes-1] – Combined weights matrix for the seed_V parameter to the fit method

n_iter_
int – The number of iterations that were run during training.

n_support_
int – The number of support vectors that were found

SVs_
array, shape = [n_observations,] – Index vector that marks the support vectors (1 = SV, 0 = no SV)

See also:

GenSVMGridSearchCV: Helper class to run an efficient grid search for GenSVM.

fit (*X*, *y*, *sample_weight=None*, *seed_V=None*)
Fit the GenSVM model on the given data

The model can be fit with or without a seed matrix (*seed_V*). This can be used to provide warm starts for the algorithm.

Parameters

- **X** (*array, shape = (n_observations, n_features)*) – The input data. It is expected that only numeric data is given.
- **y** (*array, shape = (n_observations,)*) – The label vector, labels can be numbers or strings.

- **sample_weight** (*array, shape = (n_observations,)*) – Array of weights that are assigned to individual samples. If not provided, then the weight specification in the constructor is used ('unit' or 'group').
- **seed_V** (*array, shape = (n_features+1, n_classes-1), optional*) – Seed coefficient array to use as a warm start for the optimization. It can for instance be the *combined_coef_* attribute of a different GenSVM model. This is only supported for the linear kernel.

  NOTE: the size of the seed_V matrix is *n_features+1* by *n_classes - 1*. The number of columns of *seed_V* is leading for the number of classes in the model. For example, if *y* contains 3 different classes and *seed_V* has 3 columns, we assume that there are actually 4 classes in the problem but one class is just represented in this training data. This can be useful for problems were a certain class has only a few samples.

Returns **self** – Returns self.

Return type object

predict(*X*, *trainX=None*)

Predict the class labels on the given data

Parameters

- **X** (*array, shape = [n_test_samples, n_features]*) – Data for which to predict the labels
- **trainX** (*array, shape = [n_train_samples, n_features]*) – Only for nonlinear prediction with kernels: the training data used to train the model.

Returns **y_pred** – Predicted class labels of the data in X.

Return type array, shape = (n_samples,)

6.3 GenSVMGridSearchCV

class gensvm.gridsearch.**GenSVMGridSearchCV**(*param_grid='tiny'*, *scoring=None*, *iid=True*, *cv=None*, *refit=True*, *verbose=0*, *return_train_score=True*)

GenSVM cross validated grid search

This class implements efficient GenSVM grid search with cross validation. One of the strong features of GenSVM is that seeding the classifier properly can greatly reduce total training time. This class ensures that the grid search is done in the most efficient way possible.

The implementation of this class is based on the GridSearchCV class in scikit-learn. The documentation of the various parameters is therefore mostly the same. This is done to provide the user with a familiar and easy-to-use interface to doing a grid search with GenSVM. A separate class was needed to benefit from the fast low-level C implementation of grid search in the GenSVM library.

Parameters

- **param_grid** (*string, dict, or list of dicts*) – If a string, it must be either 'tiny', 'small', or 'full' to load the predefined parameter grids (see the functions `load_grid_tiny()`, `load_grid_small()`, and `load_grid_full()`).

  Otherwise, a dictionary of parameter names (strings) as keys and lists of parameter settings to evaluate as values, or a list of such dicts. The GenSVM model will be evaluated at all combinations of the parameters.

- **scoring** (*string, callable, list/tuple, dict or None*) – A single string (see The scoring parameter: defining model evaluation rules) or a callable (see Defining your scoring strategy from metric functions) to evaluate the predictions on the test set.

  For evaluating multiple metrics, either give a list of (unique) strings or a dict with names as keys and callables as values.

  NOTE that when using custom scorers, each scorer should return a single value. Metric functions returning a list/array of values can be wrapped into multiple scorers that return one value each.

  If None, the accuracy_score is used.

- **iid** (*boolean, default=True*) – If True, the data is assumed to be identically distributed across the folds, and the loss minimized is the total loss per sample and not the mean loss across the folds.

- **cv** (*int, cross-validation generator or an iterable, optional*) – Determines the cross-validation splitting strategy. Possible inputs for cv are:

  – None, to use the default 5-fold cross validation,

  – integer, to specify the number of folds in a *(Stratified)KFold*,

  – An object to be used as a cross-validation generator.

  – An iterable yielding train, test splits.

  For integer/None inputs, `StratifiedKFold` is used. In all other cases, `KFold` is used.

  Refer to the scikit-learn User Guide on cross validation for the various strategies that can be used here.

  NOTE: At the moment, the ShuffleSplit and StratifiedShuffleSplit are not supported in this class. If you need these, you can use the GenSVM classifier directly with the GridSearchCV object from scikit-learn. (these methods require significant changes in the low-level library before they can be supported).

- **refit** (*boolean, or string, default=True*) – Refit the GenSVM estimator with the best found parameters on the whole dataset.

  For multiple metric evaluation, this needs to be a string denoting the scorer to be used to find the best parameters for refitting the estimator at the end.

  The refitted estimator is made available at the *:attr:best_estimator_ <.GenSVMGridSearchCV.best_estimator_>* attribute and allows the user to use the `predict()` method directly on this `GenSVMGridSearchCV` instance.

  Also for multiple metric evaluation, the attributes *best_index_*, *best_score_* and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer.

  See `scoring` parameter to know more about multiple metric evaluation.

- **verbose** (*integer*) – Controls the verbosity: the higher, the more messages.

- **return_train_score** (*boolean, default=True*) – If `False`, the *cv_results_* attribute will not include training scores.

Examples

```
>>> from gensvm import GenSVMGridSearchCV
>>> from sklearn.datasets import load_iris
>>> iris = load_iris()
>>> param_grid = {'p': [1.0, 2.0], 'kappa': [-0.9, 0.0, 1.0]}
>>> clf = GenSVMGridSearchCV(param_grid)
>>> clf.fit(iris.data, iris.target)
GenSVMGridSearchCV(cv=None, iid=True,
      param_grid={'p': [1.0, 2.0], 'kappa': [-0.9, 0.0, 1.0]},
      refit=True, return_train_score=True, scoring=None, verbose=0)
```

cv_results_

dict of numpy (masked) ndarrays – A dict with keys as column headers and values as columns, that can be imported into a pandas DataFrame.

For instance the below given table

param_kernel	param_gamma	param_degree	split0_test_score	...	rank_t...
'poly'	–	2	0.8	...	2
'poly'	–	3	0.7	...	4
'rbf'	0.1	–	0.8	...	3
'rbf'	0.2	–	0.9	...	1

will be represented by a `cv_results_` dict of:

```
{
'param_kernel': masked_array(data = ['poly', 'poly', 'rbf', 'rbf'],
                             mask = [False False False False]...)
'param_gamma': masked_array(data = [-- -- 0.1 0.2],
                            mask = [ True  True False False]...),
'param_degree': masked_array(data = [2.0 3.0 -- --],
                             mask = [False False  True  True]...),
'split0_test_score'  : [0.8, 0.7, 0.8, 0.9],
'split1_test_score'  : [0.82, 0.5, 0.7, 0.78],
'mean_test_score'    : [0.81, 0.60, 0.75, 0.82],
'std_test_score'     : [0.02, 0.01, 0.03, 0.03],
'rank_test_score'    : [2, 4, 3, 1],
'split0_train_score' : [0.8, 0.9, 0.7],
'split1_train_score' : [0.82, 0.5, 0.7],
'mean_train_score'   : [0.81, 0.7, 0.7],
'std_train_score'    : [0.03, 0.03, 0.04],
'mean_fit_time'      : [0.73, 0.63, 0.43, 0.49],
'std_fit_time'       : [0.01, 0.02, 0.01, 0.01],
'mean_score_time'    : [0.007, 0.06, 0.04, 0.04],
'std_score_time'     : [0.001, 0.002, 0.003, 0.005],
'params'             : [{'kernel': 'poly', 'degree': 2}, ...],
}
```

NOTE:

The key `'params'` is used to store a list of parameter settings dicts for all the parameter candidates.

The `mean_fit_time`, `std_fit_time`, `mean_score_time` and `std_score_time` are all in seconds.

For multi-metric evaluation, the scores for all the scorers are available in the *`cv_results_`* dict at the keys ending with that scorer's name (`'_<scorer_name>'`) instead of `'_score'` shown above.

('split0_test_precision', 'mean_train_precision' etc.)

best_estimator_
estimator or dict – Estimator that was chosen by the search, i.e. estimator which gave highest score (or smallest loss if specified) on the left out data. Not available if `refit=False`.

See `refit` parameter for more information on allowed values.

best_score_
float – Mean cross-validated score of the best_estimator

For multi-metric evaluation, this is present only if `refit` is specified.

best_params_
dict – Parameter setting that gave the best results on the hold out data.

For multi-metric evaluation, this is present only if `refit` is specified.

best_index_
int – The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting.

The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

For multi-metric evaluation, this is present only if `refit` is specified.

scorer_
function or a dict – Scorer function used on the held out data to choose the best parameters for the model.

For multi-metric evaluation, this attribute holds the validated `scoring` dict which maps the scorer key to the scorer callable.

n_splits_
int – The number of cross-validation splits (folds/iterations).

Notes

The parameters selected are those that maximize the score of the left out data, unless an explicit score is passed in which case it is used instead.

See also:

ParameterGrid: Generates all the combinations of a hyperparameter grid.

GenSVM: The GenSVM classifier

fit (*X*, *y*, *groups=None*)
Run GenSVM grid search with all sets of parameters

Parameters

- **X** (*array-like, shape = (n_samples, n_features)*) – Training data, where n_samples is the number of observations and n_features is the number of features.
- **y** (*array-like, shape = (n_samples,)*) – Target vector for the training data.
- **groups** (*array-like, with shape (n_samples,), optional*) – Group labels for the samples used while splitting the dataset into train/test sets.

Returns **self** – Return self.

Return type object

predict(*X*, *trainX=None*)
Predict the class labels on the test data

Parameters

- **X** (*array-like, shape = (n_samples, n_features)*) – Test data, where n_samples is the number of observations and n_features is the number of features.
- **trainX** (*array, shape = [n_train_samples, n_features]*) – Only for nonlinear prediction with kernels: the training data used to train the model.

Returns **y_pred** – Predicted class labels of the data in X.

Return type array-like, shape = (n_samples,)

score(*X*, *y*)
Compute the score on the test data given the true labels

Parameters

- **X** (*array-like, shape = (n_samples, n_features)*) – Test data, where n_samples is the number of observations and n_features is the number of features.
- **y** (*array-like, shape = (n_samples,)*) – True labels for the test data.

Returns **score**

Return type float

6.4 Parameter Grids

gensvm.gridsearch.**load_grid_tiny**()
Load a tiny parameter grid for the GenSVM grid search

This function returns a parameter grid to use in the GenSVM grid search. This grid was obtained by analyzing the experiments done for the GenSVM paper and selecting the configurations that achieve accuracy within the 95th percentile on over 90% of the datasets. It is a good start for a parameter search with a reasonably high chance of achieving good performance on most datasets.

Note that this grid is only tested to work well in combination with the linear kernel.

Returns **pg** – List of 10 parameter configurations that are likely to perform reasonably well.

Return type list

gensvm.gridsearch.**load_grid_small**()
Load a small parameter grid for GenSVM

This function loads a default parameter grid to use for the #' GenSVM gridsearch. It contains all possible combinations of the following #' parameter sets:

```
pg = {
    'p': [1.0, 1.5, 2.0],
    'lmd': [1e-8, 1e-6, 1e-4, 1e-2, 1],
    'kappa': [-0.9, 0.5, 5.0],
    'weights': ['unit', 'group'],
}
```

Returns **pg** – Mapping from parameters to lists of values for those parameters. To be used as input for the `GenSVMGridSearchCV` class.

Return type dict

gensvm.gridsearch.**load_grid_full**()

Load the full parameter grid for GenSVM

This is the parameter grid used in the GenSVM paper to run the grid search experiments. It uses a large grid for the `lmd` regularization parameter and converges with a stopping criterion of `1e-8`. This is a relatively small stopping criterion and in practice good classification results can be obtained by using a larger stopping criterion.

The function returns the following grid:

```
pg = {
        'lmd': [pow(2, x) for x in range(-18, 19, 2)],
        'kappa': [-0.9, 0.5, 5.0],
        'p': [1.0, 1.5, 2.0],
        'weights': ['unit', 'group'],
        'epsilon': [1e-8],
        'kernel': ['linear']
     }
```

Returns **pg** – Mapping from parameters to lists of values for those parameters. To be used as input for the `GenSVMGridSearchCV` class.

Return type dict

6.5 Kernels in GenSVM

Kernels in GenSVM are implemented as follows.

- Radial Basis Function (RBF):

$$k(x_1, x_2) = \exp(-\gamma \|x_1 - x_2\|^2)$$

- Polynomial:

$$k(x_1, x_2) = (\gamma x_1' x_2 + coef)^{degree}$$

- Sigmoid:

$$k(x_1, x_2) = \tanh(\gamma x_1' x_2 + coef)$$

6.6 Change Log

6.6.1 Version 0.2.7

- Deal with various deprecated features of sklearn

6.6.2 Version 0.2.6

- Minor fixes to fix description on PyPI

6.6.3 Version 0.2.5

- Build platform wheels for Linux and MacOS
- Minor improvements to the package

6.6.4 Version 0.2.4

- Add support for retrieving support vectors

6.6.5 Version 0.2.3

- Bugfix for prediction with gamma = 'auto'

6.6.6 Version 0.2.2

- Add error when unsupported ShuffleSplits are used

6.6.7 Version 0.2.1

- Update docs
- Speed up unit tests

6.6.8 Version 0.2.0

- Add support for interrupting training and retreiving partial results
- Allow specification of sample weights in GenSVM.fit()
- Extract per-split durations from grid search results
- Add pre-defined parameter grids 'tiny', 'small', and 'full'
- Add code for prediction with kernels
- Add unit tests
- Change default coef in poly kernel to 1.0 for inhomogeneous kernel
- Minor bugfixes, documentation improvement, and code cleanup
- Add continuous integration through Travis-CI.

6.6.9 Version 0.1.6

- Fix segfault caused by error function in C library.
- Add "load_default_grid" function to gensvm.gridsearch

Index