SmartSVM Documentation

Release "1.1.0"

Gertjan van den Burg

Jun 11, 2019

Contents

1 Installation **3**

2 Usage **5**
- 2.1 Citing . . . 5
- 2.2 Bayes error estimates . . . 5
- 2.3 SmartSVM Classifier . . . 6

3 Known Limitations **9**

4 References **11**

5 API documentation **13**
- 5.1 smartsvm.base module . . . 13
- 5.2 smartsvm.cross_connections module . . . 14
- 5.3 smartsvm.cut module . . . 15
- 5.4 smartsvm.divergence module . . . 16
- 5.5 smartsvm.error_estimate module . . . 17
- 5.6 smartsvm.smartsvm module . . . 19
- 5.7 smartsvm.utils module . . . 20

Python Module Index **21**

Index **23**

SmartSVM is a Python package that implements the methods from Fast Meta-Learning for Adaptive Hierarchical Classifier Design by Gerrit J.J. van den Burg and Alfred O. Hero. The package contains functions for estimating the Bayes error rate (BER) using the Henze-Penrose divergence and a hierarchical classifier called SmartSVM. See the Usage documentation below for more details.

CHAPTER 1

Installation

SmartSVM is available on PyPI and can be installed easily with:

```
pip install smartsvm
```

CHAPTER 2

Usage

In the paper the main focus is on the accurate Bayes error estimates and the hierarchical classifier SmartSVM. These will therefore be of most interest to users of the SmartSVM package. Below we briefly explain how to use these functions.

2.1 Citing

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

```bibtex
@article{van2017fast,
  title={Fast Meta-Learning for Adaptive Hierarchical Classifier Design},
  author={Gerrit J.J. van den Burg and Alfred O. Hero},
  journal={arXiv preprint arXiv:1711.03512},
  archiveprefix={arXiv},
  year={2017},
  eprint={1711.03512},
  url={https://arxiv.org/abs/1711.03512},
  primaryclass={cs.LG}
}
```

2.2 Bayes error estimates

Error estimation is implemented three functions:

- `hp_estimate` for the Henze-Penrose estimator of the Bayes error rate. This can be used as:

```python
>>> import numpy as np
>>> from smartsvm import hp_estimate
>>> X1 = np.random.multivariate_normal([-1, 0], [[1, 0], [0, 1]], 100)
>>> X2 = np.random.multivariate_normal([1, 0], [[1, 0], [0, 1]], 100)
```

(continues on next page)

(continued from previous page)

```
>>> hp_estimate(X1, X2) # with normalization
>>> hp_estimate(X1, X2, normalize=False) # without normalization
```

- `compute_error_graph` and `compute_ovr_error` respectively compute the complete weighted graph of pairwise BER estimates or the One-vs-Rest BER for each class. They have a similar interface:

```
>>> import numpy as np
>>> from smartsvm import compute_error_graph, compute_ovr_error
>>> from sklearn.datasets import load_digits
>>> digits = load_digits(5)
>>> n_samples = len(digits.images)
>>> X = digits.images.reshape((n_samples, -1))
>>> y = digits.target
>>> G = compute_error_graph(X, y, n_jobs=2, normalize=True)
>>> d = compute_ovr_error(X, y, normalize=True)
```

2.3 SmartSVM Classifier

SmartSVM is an adaptive hierarchical classifier which constructs a classification hierarchy based on the Henze-Penrose estimates of the Bayes error between each pair of classes. The classifier is build on top of Scikit-Learn and can be used in the exact same way as other sklearn classifiers:

```
>>> import numpy as np
>>> from smartsvm import SmartSVM
>>> from sklearn.datasets import load_digits
>>> digits = load_digits(10)
>>> n_samples = len(digits.images)
>>> X = digits.images.reshape((n_samples, -1))
>>> y = digits.target
>>> clf = SmartSVM()
>>> clf.fit(X, y)
>>> clf.predict(X)
```

By default, the SmartSVM classifier uses the Linear Support Vector Machine (`LinearSVC`) as the underlying binary classifier for each binary subproblem in the hierarchy. This can easily be changed with the `binary_clf` parameter to the class constructor, for instance:

```
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = SmartSVM(binary_clf=DecisionTreeClassifier)
>>> clf.fit(X, y)
>>> clf._get_binary()
DecisionTreeClassifier(class_weight=None, criterion='gini',
        max_depth=None, max_features=None, max_leaf_nodes=None,
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, presort=False, random_state=None,
        splitter='best')
```

You may optionally add parameters for the classifier through the `clf_params` parameter. This should be a dict with the parameters to the binary classifier, as follows:

```
>>> clf = SmartSVM(binary_clf=DecisionTreeClassifier, clf_params={'criterion':
↪'entropy'})
>>> clf.fit(X, y)
```

(continues on next page)

(continued from previous page)

```
>>> clf._get_binary()
DecisionTreeClassifier(class_weight=None, criterion='entropy',
        max_depth=None, max_features=None, max_leaf_nodes=None,
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, presort=False, random_state=None,
        splitter='best')
```

Finally, it's possible to retrieve probability estimates for the classes if the underlying classifier supports the `predict_proba` method:

```
>>> from sklearn.svm import SVC
>>> clf = SmartSVM(binary_clf=SVC, clf_params={"probability": True})
>>> clf.fit(X, y)
>>> prob = clf.predict_proba(X)
>>> import pandas as pd
>>> df = pd.DataFrame(prob)
>>> df
                 0             1             2      ...
0     9.999997e-01  1.716831e-18  2.677824e-13      ...
1     1.000000e-07  9.956408e-01  1.035589e-09      ...
2     2.595652e-05  1.452011e-02  9.722321e-01      ...
```

For more information about parameters to SmartSVM, see the API documentation here.

CHAPTER 3

Known Limitations

The Henze-Penrose estimator of the Bayes error rate is based on construction of the Euclidean minimal spanning tree. The current algorithm for this in the SmartSVM package uses an adaptation of Whitney's algorithm. This is not the fastest way to construct a minimal spanning tree. The Fast Euclidean Minimal Spanning Tree algorithm by March et al., would be a faster option but this makes it more difficult to construct orthogonal MSTs. Incorporating this algorithm into the SmartSVM package is considered a topic for future work.

CHAPTER 4

References

The main reference for this package is:

- G.J.J. van den Burg and A.O. Hero - Fast Meta-Learning for Adaptive Hierarchical Classifier Design (2017).

The theory of the Henze-Penrose estimator is developed in:

- V. Berisha, A. Wisler, A.O. Hero, A. Spanias - Empirically Estimable Classification Bounds Based on a Non-parametric Divergence Measure (2016).
- V. Berisha, A.O. Hero - Empirical Non-Parametric Estimation of the Fisher Information (2015).

CHAPTER 5

API documentation

The documentation below is automatically generated from the docstrings in the code. It is therefore also available through the builtin Python `help()` function.

5.1 smartsvm.base module

Base object for hierarchical classifiers

This module contains the definition of a general hierarchical classifier, which forms the basis for the *SmartSVM* classifier. The *HierarchicalClassifier* class is an abstract base class, which leaves the `fit` and `_split` methods to be defined in the subclass.

class smartsvm.base.**HierarchicalClassifier**(*binary_clf=<class 'sklearn.svm.classes.LinearSVC'>*, *clf_params=None*, *n_jobs=1*)

Bases: `sklearn.base.BaseEstimator`, `sklearn.base.ClassifierMixin`

Base class for a hierarchical classifier

This class is a base class for a hierarchical classifier which contains a hierarchy of binary classification problems. It forms the basis of the *SmartSVM* class and can be used to implement other hierarchical classifiers. Note that this class is also the node class for the binary tree: it has children which are in turn also *HierarchicalClassifier* instances.

Parameters

- **binary_clf** (*classifier*) – The type of the binary classifier to be used for each binary subproblem. This will typically be a scikit-learn classifier such as LinearSVC, DecisionTreeClassifier, etc.
- **clf_params** (*dict*) – Parameters to pass on to the constructor of the `binary_clf` type classifier. It must be a dict with a mapping from parameter to value.
- **n_jobs** (*int*) – Number of parallel jobs to use where applicable.

classifier_
The binary classifier for this node in the tree, if it exists. This can also be obtained with the `_get_binary()` method.

Type classifier

negative_child_
The "left" child of the binary tree below this node.

Type *HierarchicalClassifier*

positive_child_
The "right" child of the binary tree below this node.

Type *HierarchicalClassifier*

negative_
Set of labels of the binary subproblem that are on the "left" part of the tree.

Type set

positive_
Set of labels on the binary subproblem that are on the "right" part of the tree.

Type set

predict(*X*)
Perform classification on dataset X

elements
Elements in the graph or list of nodes

fit(*X*, *y*)
Fit model.

is_splitable
Check if the elements of this node can be split further

is_splitted
Check if this node is already split

predict(*X*)
Predict the class labels using the hierarchical classifier

predict_proba(*X*)
Predict the class probabilities using the hierarchical classifier.

This is only available if the underlying binary classifier has the "predict_proba" method. Note that if the probability model is created using cross validation, the results can be slightly different than those obtained with the predict method.

print_tree(*prefix=''*, *is_tail=None*)
Print the tree

5.2 smartsvm.cross_connections module

Code for computing the number of cross connections in an MST.

The functions in this module are used to compute the number of cross-connections between data points of different classes in the Euclidean Minimal Spanning Tree. It is the interface between the Cython code talking to the C implementations, and the higher level functions that compute the Henze-Penrose divergence.

`smartsvm.cross_connections.`**`binary_cross_connections`**(*X*, *y*, *nTrees=3*)

Compute the number of cross connections for two classes

This function computes the average number of non-trivial cross-connections in the orthogonal MSTs between points from different classes. For each MST the number of times a point from one class is connected to a point from a different class is recorded. This is reduced by 1 to adjust for the trivial connection that will always exist, even in the case of large separation. The result is averaged for each of the orthogonal MSTs.

A warning can occur when the requested number of orthogonal MSTs can't be computed on the data. This happens when there are either too few datapoints to construct this many MSTs, or in the extreme case where each edge to a data point is used in previous MSTs.

Parameters

- **X** (*numpy.ndarray*) – Data matrix
- **y** (*np.ndarray*) – Class labels, assumed to be only +1 and -1
- **nTrees** (*int*) – The number of orthogonal MSTs to compute

Returns **C** – The (average) number of non-trivial connections between instances of different classes in the MSTs

Return type float

`smartsvm.cross_connections.`**`ovr_cross_connections`**(*X*, *y*, *nClass*, *nTrees=3*)

Compute the One-vs-Rest cross-connection counts

This function computes the cross-connection counts in the One-vs-Rest setting by constructing orthogonal MSTs on the entire dataset, and collecting cross-connection counts for each class.

Parameters

- **X** (*numpy.ndarray*) – Numpy array of the data matrix
- **y** (*numpy.ndarray*) – Numpy array of the class labels, assumed to be in 0..nClass-1
- **nClass** (*int*) – The number of classes in y for the full problem. This needs to be supplied in case y is a subset of the full dataset where not all classes are present. This ensures that the outcome matrix has the appropriate size.
- **nTrees** (*int*) – The number of orthogonal MSTs to compute

Returns **Cs** – Vector of class cross-connection counts, ordered by class label (nClass x nClass)

Return type numpy.ndarray

5.3 smartsvm.cut module

Functions for graph cutting

This module contains functions for cutting the weighted error graph. The default algorithm for cutting the graph is the Stoer-Wagner algorithm, but this module also contains code for experimenting with the Spectral Clustering algorithm to create graph cuts.

`smartsvm.cut.`**`graph_cut`**(*G*, *algorithm='stoer_wagner'*)

Cut a connected graph into two disjoint graphs

This is a wrapper function to make it easy to use different graph cut algorithms.

Parameters

- **G** (*networkx.Graph*) – The graph that needs to be cut

- **algorithm** (*str*) – The graph cut algorithm to use. Available values are: `'stoer_wagner'`, `'spectral_clustering'`, and `'normalized_cut'`.

Returns

- **G1** (*networkx.Graph*) – One of the subgraphs
- **G2** (*networkx.Graph*) – The other subgraph

Raises `ValueError` – When an unknown graph cut algorithm is supplied.

`smartsvm.cut.`**`graph_cut_sc`**(*G*, *assign_labels=None*)

Apply the Spectral Clustering algorithm to cut the graph

Create two subgraphs using a Spectral Clustering of the adjacency matrix of the graph.

Important: Since the matrix of weights is a dissimilarity matrix (high numbers correspond to difficult to separate classes, we turn it into a similarity matrix for the Spectral Clustering algorithm by using the normalized exponent of the weight matrix. This is also done in the examples of Scikit-Learn for Spectral Clustering. Weights that were zero in the weight matrix are set to zero in the dissimilarity matrix as well.

Parameters

- **G** (*networkx.Graph*) – The graph that needs to be cut
- **assign_labels** (*str*) – Parameter for the Scikit-Learn `spectral_clustering` function. Available values are: `kmeans` and `discretize`.

Returns

- **G1** (*networkx.Graph*) – One of the subgraphs
- **G2** (*networkx.Graph*) – The other subgraph

`smartsvm.cut.`**`graph_cut_sw`**(*G*)

Apply the Stoer-Wagner graph cut

Use the Stoer-Wagner algorithm for cutting the graph.

Parameters **G** (*networkx.Graph*) – The graph that needs to be cut

Returns

- **G1** (*networkx.Graph*) – One of the subgraphs
- **G2** (*networkx.Graph*) – The other subgraph

5.4 smartsvm.divergence module

Functions for computing the Henze-Penrose divergence

This module contains functions for computing the Henze-Penrose divergence between data from two classes (*divergence()*) or between one class and the collection of all other classes (*ovr_divergence()*).

`smartsvm.divergence.`**`divergence`**(*X1*, *X2*, *nTrees=3*, *bias_correction=True*)

Compute the Henze-Penrose divergence

Compute the Henze-Penrose divergence between data from two classes using the Friedman-Rafsky statistic. This is based on the Euclidean minimal spanning tree between two classes. To reduce variance in the estimate, a number of orthogonal MSTs are used that can be set with a function parameter. A bias correction to the divergence is applied, unless this is disabled by the user (if disabled, the estimate is still corrected to ensure non-negativity).

Parameters

- **X1** (*numpy.ndarray*) – Data matrix for one of the classes
- **X2** (*numpy.ndarray*) – Data matrix for the other class
- **nTrees** (*int*) – Number of orthogonal minimal spanning trees to use
- **bias_correction** (*bool*) – Whether or not to apply bias correction to the estimator

Returns **divergence** – The Henze-Penrose divergence between the classes

Return type float

`smartsvm.divergence.`**`merge_and_label`**(*X1*, *X2*)

Merge two datasets

Merge the datasets into one array and create a vector of labels to preserve class membership.

Parameters

- **X1** (*numpy.ndarray*) – Data matrix for one of the classes
- **X2** (*numpy.ndarray*) – Data matrix for the other class

Returns

- **data** (*numpy.ndarray*) – Data matrix which vertically stacks X1 and X2
- **labels** (*numpy.ndarray*) – Labels vector of -1 and 1 for X1 and X2 data respectively

`smartsvm.divergence.`**`ovr_divergence`**(*X*, *labels*, *nTrees=3*, *bias_correction=True*)

Compute the One-vs-Rest Henze-Penrose Divergence for all classes

This function is similar to `divergence()` with the difference that it computes the One-vs-Rest divergence. This is the divergence between one class and the collection of all the other classes together. This divergence is computed for each class simultaneously.

Parameters

- **X** (*numpy.ndarray*) – Data matrix
- **labels** (*numpy.ndarray*) – Class labels for the instances
- **nTrees** (*int*) – The number of orthogonal MSTs to construct
- **bias_correction** (*bool*) – Whether or not to apply bias correction

Returns **ovr_divergence** – Dictionary with a mapping from the class label to the OvR divergence estimate

Return type dict

5.5 smartsvm.error_estimate module

Functions for computing the Henze-Penrose estimate of the Bayes Error Rate

This module contains functions for computing the Henze-Penrose estimate of the Bayes Error Rate (BER).

`smartsvm.error_estimate.`**`compute_error_graph`**(*X*, *y*, *n_jobs=1*, *normalize=True*)

Compute the complete weighted graph of pairwise BER estimates

Computes the estimate of the BER for each pair of classes and returns this in a complete weighted graph. If desired, computing the error can be done in parallel.

Parameters

- **X** (*numpy.ndarray*) – Numpy array of the data matrix

- **y** (*numpy.ndarray*) – Numpy array with the class labels
- **n_jobs** (*int*) – Number of parallel jobs to use
- **normalize** (*bool*) – Whether or not to use normalized BER estimation

Returns **G** – Weighted complete graph where the class labels are the nodes and the edge weights are given by the BER estimate

Return type networkx.Graph

smartsvm.error_estimate.**compute_ovr_error**(*X*, *y*, *normalize=True*)

Compute OvR-BER for each class

The One-vs-Rest Bayes error rate is the error rate for a single class compared to all other classes combined. This function computes this error rate for each class in the dataset, using the *divergence.ovr_divergence()* function. By default the OvR-BER is normalized with the estimates of the class prior probabilities.

Parameters

- **X** (*numpy.ndarray*) – The data matrix
- **y** (*numpy.ndarray*) – A vector of class labels
- **normalize** (*bool*) – Whether or not to normalize the OvR-BER

Returns **estimates** – Dictionary with a mapping from the class label to a float representing the OvR-BER estimate.

Return type dict

smartsvm.error_estimate.**hp_binary**(*X*, *y*, *normalize=True*)

Henze-Penrose estimation of the BER for a binary dataset

This is a wrapper around the *hp_estimate()* function for binary datasets.

Parameters

- **X** (*numpy.ndarray*) – Data matrix
- **y** (*numpy.ndarray*) – Label array consisting of two distinct labels
- **normalize** (*bool*) – Whether or not to normalize the error using empirical estimates of prior probabilities

Returns **estimate** – The Henze-Penrose estimate of the Bayes error rate

Return type float

smartsvm.error_estimate.**hp_estimate**(*X1*, *X2*, *normalize=True*)

Henze-Penrose estimation of the Bayes Error Rate

Estimate the (normalized) Bayes error rate using the Henze-Penrose estimator. The estimate is formed by averaging the upper and lower bounds on the Bayes error. By default, the estimate is normalized with the empirical estimates of the class prior probability.

Parameters

- **X1** (*numpy.ndarray*) – Data matrix for one class
- **X2** (*numpy.ndarray*) – Data matrix for the other class
- **normalize** (*bool*) – Whether or not to normalize the error using empirical estimates of prior probabilities

Returns **estimate** – The Henze-Penrose estimate of the Bayes error rate

Return type float

5.6 smartsvm.smartsvm module

class smartsvm.smartsvm.**SmartSVM**(*binary_clf=<class 'sklearn.svm.classes.LinearSVC'>, clf_params=None, n_jobs=1, graph=None, cut_algorithm='stoer_wagner', normalize_error=True*)

Bases: *smartsvm.base.HierarchicalClassifier*

SmartSVM classifier for multiclass classification

This is the SmartSVM classifier. This classifier splits the classes into a hierarchy based on the weighted complete graph of pairwise estimates of the Bayes Error Rate between classes.

Note that this class is used in a binary tree. It therefore has children which are also elements of the *SmartSVM* class.

This class inherits from the *HierarchicalClassifier* class, more documentation on the basic methods of the class can be found there.

Parameters

- **binary_clf** (*classifier*) – The type of the binary classifier to be used for each binary subproblem. This will typically be a scikit-learn classifier such as LinearSVC, DecisionTreeClassifier, etc.
- **clf_params** (*dict*) – Parameters to pass on to the constructor of the `binary_clf` type classifier. It must be a dict with a mapping from parameter to value.
- **n_jobs** (*int*) – Number of parallel jobs to use where applicable.
- **graph** (*networkx.Graph*) – Complete weighted graph with the pairwise Bayes error estimates. If it is not supplied, it will be computed when the *fit()* method is called. This parameter exists for situations when the graph is precomputed.
- **cut_algorithm** (*str*) – The algorithm to use for the graph cut. Several algorithms are currently implemented, see *cut.graph_cut()* for more details.
- **normalize_error** (*bool*) – Whether or not to normalize the Bayes error estimates with the empirical estimate of the prior probability.

elements

The labels of this node in the hierarchy

fit(*X, y*)

Fit the SmartSVM classifier

This method fits the SmartSVM classifier on the given data. If the `graph` attribute of the class is not defined, it will be computed with *error_estimate.compute_error_graph()*. If this node in the hierarchy can not be split further, no classifier will be fit. Otherwise, the node will be split to form a binary classification problem. Next, the binary classifier will be trained on this problem. Finally, the left and right children of the classifier will be trained as a recursive step.

Important: This method constructs the `graph` attribute of the class if it does not exist yet. However, if the graph does exist, it is not recomputed. This means that a problem can occur if you construct an instance of the *SmartSVM* classifier, fit it, then fit it again with different data. Namely, the graph for the first data may not be appropriate for the second dataset. The solution is however simple: when fitting with a different dataset, reset the graph using `clf.graph = None`.

Parameters

- **X** (*numpy.ndarray, accept csr sparse*) – Training data, feature matrix
- **y** (*numpy.ndarray*) – Training data, label vector

Returns **self** – Returns self.

Return type object

5.7 smartsvm.utils module

Useful numerical utilities.

`smartsvm.utils.`**`indices_from_classes`**(*classes*, *y*)

Create index vector for all labels in classes

Create an index vector with the same dimensions as the vector y, with indices equal to True if the label is in the list of classes.

Parameters

- **`classes`** (*`list or numpy array`*) – Classes to check membership of
- **`y`** (*`numpy.ndarray`*) – Label vector

Returns **indices** – Boolean vector with True for the elements of `y` which occur in `classes` and False otherwise.

Return type numpy.ndarray (bool)

Python Module Index

S

smartsvm.base, 13
smartsvm.cross_connections, 14
smartsvm.cut, 15
smartsvm.divergence, 16
smartsvm.error_estimate, 17
smartsvm.smartsvm, 19
smartsvm.utils, 20

Index

B

`binary_cross_connections()` (*in module smartsvm.cross_connections*), 14

C

`classifier_` (*smartsvm.base.HierarchicalClassifier attribute*), 13
`compute_error_graph()` (*in module smartsvm.error_estimate*), 17
`compute_ovr_error()` (*in module smartsvm.error_estimate*), 18

D

`divergence()` (*in module smartsvm.divergence*), 16

E

`elements` (*smartsvm.base.HierarchicalClassifier attribute*), 14
`elements` (*smartsvm.smartsvm.SmartSVM attribute*), 19

F

`fit()` (*smartsvm.base.HierarchicalClassifier method*), 14
`fit()` (*smartsvm.smartsvm.SmartSVM method*), 19

G

`graph_cut()` (*in module smartsvm.cut*), 15
`graph_cut_sc()` (*in module smartsvm.cut*), 16
`graph_cut_sw()` (*in module smartsvm.cut*), 16

H

`HierarchicalClassifier` (*class in smartsvm.base*), 13
`hp_binary()` (*in module smartsvm.error_estimate*), 18
`hp_estimate()` (*in module smartsvm.error_estimate*), 18

I

`indices_from_classes()` (*in module smartsvm.utils*), 20
`is_splitable` (*smartsvm.base.HierarchicalClassifier attribute*), 14
`is_splitted` (*smartsvm.base.HierarchicalClassifier attribute*), 14

M

`merge_and_label()` (*in module smartsvm.divergence*), 17

N

`negative_` (*smartsvm.base.HierarchicalClassifier attribute*), 14
`negative_child_` (*smartsvm.base.HierarchicalClassifier attribute*), 14

O

`ovr_cross_connections()` (*in module smartsvm.cross_connections*), 15
`ovr_divergence()` (*in module smartsvm.divergence*), 17

P

`positive_` (*smartsvm.base.HierarchicalClassifier attribute*), 14
`positive_child_` (*smartsvm.base.HierarchicalClassifier attribute*), 14
`predict()` (*smartsvm.base.HierarchicalClassifier method*), 14
`predict_proba()` (*smartsvm.base.HierarchicalClassifier method*), 14
`print_tree()` (*smartsvm.base.HierarchicalClassifier method*), 14

S

`SmartSVM` (*class in smartsvm.smartsvm*), 19
`smartsvm.base` (*module*), 13

`smartsvm.cross_connections` (*module*), 14
`smartsvm.cut` (*module*), 15
`smartsvm.divergence` (*module*), 16
`smartsvm.error_estimate` (*module*), 17
`smartsvm.smartsvm` (*module*), 19
`smartsvm.utils` (*module*), 20