s3l Documentation

Release 0.1.0

XXX

May 28, 2019

Contents

S3L is a python toolkit based on safe semi-supervised learning, which is a novel branch of semi-supervised learning.

Semi-supervised learning (SSL) concerns the problem on how to improve learning performance via the usage of a small amount of labeled data and a large amount of unlabeled data. Many SSL methods have been developed, e.g., generative model, graph-based method, disagreement-based method and semi-supervised SVMs. Despite the success of SSL, however, a considerable amount of empirical studies reveal that SSL with the exploitation of unlabeled data might even deteriorate learning performance. It is highly desirable to study **safe** SSL scheme that on one side could often improve performance, on the other side will not hurt performance, since the users of SSL wont expect that SSL with the usage of more data performs worse than certain direct supervised learning with only labeled data.

Specifically, *Safe*, here means that the generalization performance is never *statistically significantly worse* than methods using only labeled data.

S3L implements multiple front-end safe semi-supervised learning algorithms, and provides a weakly-supervised learning experiment framework including some well-defined protocols for learning algorithms, experiments and evaluation metrics. With this toolkit, you can build up your comparing experiments between learning algorithms with different learning settings like supervised, semi/weakly-supervised, as well as different tasks such as single/multi-label learning. We hope this toolkit could help you explore the classic semi-supervised learning algorithms and go further to test your ones.

Submit bugs or suggestions in the Issues section or feel free to submit your contributions as a pull request.

CHAPTER 1

Table of Content

1.1 Introduction

1.1.1 Functionality Overview

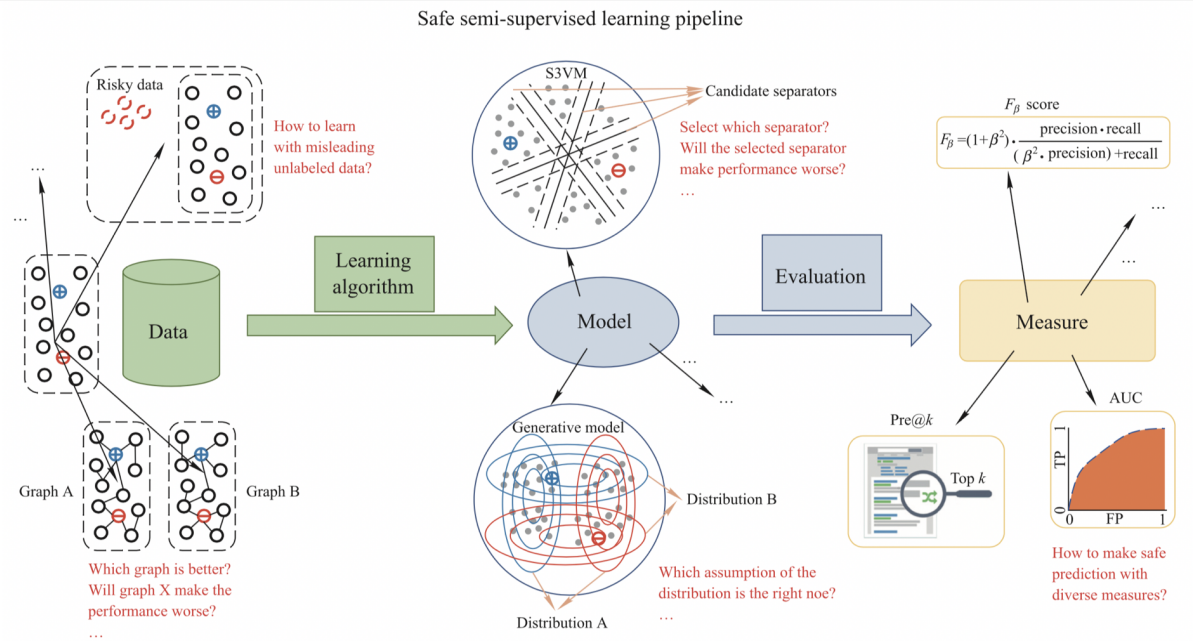
This toolkit defines a unified experimnts framework for semi-supervised learning algorithms. Besides, it also provides some state-of-the-art safe semi-supervised learning (Safe SSL) algorithms handling bad data quality and model uncertainty. We design the framework under as less as assumption except for supervised setting. Therefore, it's convenient to incorporate different algorithms for different settings into our experiment framework. We hope this unified framework can help researchers and other users evaluate machine learning algorithms in a light manner.

Pipeline of Safe SSL

In this toolkit, we foucus on three critical aspects to improve the safeness of semi-supervised learning: **data quality**, **model uncertainty** and **measure diversity**.

1. As for data quality, the graph used in graph-based SSL and risky unlabeled samples may degenerate the performance.
2. In the model part, we now understand that the exploitation of unlabeled data naturally leads to more than one model option, and inadequate choice may lead to poor performance.
3. In practical applications, the performance measures are often diverse, so the safeness should also be considered under different measures.

The figure below provides an illustration of the three aspects of the safeness problem in semi-supervised learning.



Semi-supervised learning pipeline that displays the sources of unsafeness: data quality, model uncertainty, and measure diversity

Then, we will introduce the corresponding algorithms in detail in *data quality* module, *model uncertainty* module and *ensemble* module.

For more details about *safe* semi-supervised learning, we recommend users to read Safe semi-supervised learning: a brief introduction.

1.1.2 Packages and Modules

Here, we will give a brief introduction of the submodules in this package.

- **Classification**: Classical semi-supervised learning algorithms.
- **Data Quality**: Algorithms to solve the safeness of graph-based SSL algorithms.
- **Model Uncertainty**: Algorithms to eliminate the uncertainty of classifiers.
- **Ensemble**: Ensemble methods to provide a safer prediction when given a set of training models or prediction results.
- **Experiments**: A class which designs experiment process.
- **Estimator**: A class of machine learning algorithms.
- **Metric**: Metric functions used to evaluate the prediction given ground-truth.
- **Wrapper**: Helper classes to wrap the third-party packages into the experiments in this package.

1.2 Tutorial of S3L

This documentation introduces the S3L. Examples are provided to show the generic usage under different settings. To master S3L, a careful reading of *Generic Processes th Use S3L* and further reading of *User Guide* will be much helpful.

1.2.1 Quick Start

S3L is a python package of Safe Semi-Supervised Learning.

Dependencies

The package is developed with Python3 (version 3.6|3.7 are tested in both `windows` and `linux` system).

Basic Dependencies

```
numpy >= 1.15.1
scipy >= 1.1.0
scikit-learn >= 0.19.2
cvxopt >= 1.2.0
```

Setup

You can get s3l simply by:

```
$ pip install s3l
```

Or clone s3l source code to your local directory and build from source:

```
$ cd S3L
$ python setup.py s3l
$ pip install dist/*.whl
```

Both ways would install the dependent packages with *pip* command automatically.

A Quick Example

We can use s3l for different experiments. The following example shows a possible way to do experiments based on built-in algorithms and data sets:

```
import sys, os
from s3l.Experiments import SslExperimentsWithoutGraph
from s3l.model_uncertainty.S4VM import S4VM

# algorithm configs
configs = [
        ('S4VM', S4VM(), {
            'kernel': 'RBF',
            'gamma':[0],
            'C1': [50,100],
            'C2': [0.05,0.1],
            'sample_time':[100]
        })
    ]

# datasets
# name,feature_file,label_file,split_path,graph_file
datasets = [
    ('house', None, None, None, None),
    ('isolet', None, None, None, None)
    ]

# experiments
experiments = SslExperimentsWithoutGraph(transductive=True, n_jobs=4)
experiments.append_configs(configs)
experiments.append_datasets(datasets)
experiments.set_metric(performance_metric='accuracy_score')

results = experiments.experiments_on_datasets(unlabel_ratio=0.75,test_ratio=0.2,
    number_init=2)
```

1.2.2 Generic Processes to Use S3L

We will show you two generic processes to start using S3L, which includes two parts:

1. Experiment Framework
2. Call Algorithms Directly

Experiment Framework

We provide built-in experiment process for different semi-supervised settings with different input data such as inductive/transductive, WithGraph/WithoutGraph, givenDataSplit/randomlySplit and so on. The experiment class implements the following process: `load data`, `data split`, `hyper-parameters search` and *evaluate the selected model in testing data*. In order to accelerate the experiments, we also include multi-process with `joblib`. The experiment framework allow you to evaluate supervised/semi-supervised learning algorithms in less than ten statements. Take an example,

```
import sys
import os

from s3l.Experiments import SslExperimentsWithGraph
from s3l.classification.LPA import LPA


if __name__ == '__main__':
    configs = [
        ('LPA', LPA(), {
            'kernel': ['rbf'],
            'n_neighbors':[3,5,7]
        })
    ]

    datasets = [
        ('ionosphere', None, None, None, None)
    ]
    # (name, feature_file, label_file, split_path, graph_file)

    experiments = SslExperimentsWithGraph(n_jobs=1)

    experiments.append_configs(configs)
    experiments.append_datasets(datasets)
    experiments.set_metric(performance_metric='accuracy_score')

    results = experiments.experiments_on_datasets(
        unlabel_ratio=0.75, test_ratio=0.2, number_init=4)

    # do something with results #
```

The above codes evaluates `Label Propagation` algorithm on the built-in dataset `ionosphere`. The best model is searched with `rbf kernel` and `n_neighbors` is in the range of [3, 5, 7]. Finally, the accuracy_score is reported in the local variable `result`.

Call Algorithms Directly

The built-in algorithms can be called directly as in `sklearn` package. The algorithms we have implemented are listed here. As long as reading the examples of certain algorithm in its module page, you can easily try out semi-supervised algorithm as you like. For example,

```
import sys
import os
```

(continues on next page)

(continued from previous page)

```
import numpy as np
from s3l.classification.TSVM import TSVM
from s3l.metrics.performance import accuracy_score
from s3l.datasets import base, data_manipulate


if __name__ == '__main__':
    datasets = [
        ('house', None, None),
    ]
    for name, feature_file, label_file in datasets:
        # load dataset
        X, y = base.load_dataset(name, feature_file, label_file)

        # split
        _, _, labeled_idxs, unlabeled_idxs = \
            data_manipulate.inductive_split(X=X, y=y, test_ratio=0.,
                            initial_label_rate=1 - unlabel_ratio,
                            split_count=1, all_class=True)

        labeled_idx = labeled_idxs[0]
        unlabeled_idx = unlabeled_idxs[0]

        tsvm = TSVM()
        tsvm.fit(X, y, labeled_idx)
        pred = lead.predict(X[unlabeled_idx])
        print("Accuracy_score: {}".format(
                    accuracy_score(y[unlabeled_idx], pred)))
```

The above code runs `TSVM` (Transductive Support Vector Machine) with default hyper-parameter settings given feature `X`, label `y` and indexes of labeled data‘‘labeled_idx‘‘. Then, the prediction is evaluated with accuracy score on unlabeled data.

1.3 APIs

The following links contain detailed information about every function, class and module.

1.3.1 Subpackages

classification

CoTraining

class s3l.classification.CoTraining.**CoTraining**(*pos=1*, *neg=1*, *model1=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001, verbose=False)*, *model2=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0, decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf', max_iter=-1, probability=True, random_state=None, shrinking=True, tol=0.001, verbose=False)*, *ind1=array([], dtype=int64)*, *ind2=array([], dtype=int64)*, *nepo=40*, *buffer_size=200*)

Bases: *s3l.base.InductiveEstimatorWOGraph*

CoTraining classifier

Parameters

- **pos** (*int (default=1)*) – The number of positive samples selected in each contraining iteration
- **neg** (*int (default=1)*) – The number of negative samples selected in each contraining iteration
- **model** (*object*) – [model1,model2] initializing model1 for view1 and model2 for view2
- **ind1** (*array-like [] (default=0)*) – The column index of view1 in features X
- **ind2** (*array-like [] (default=0)*) – The column index of view2 in features X
- **nepo** (*int (default=40)*) – The number of iteration
- **buffer_size** (*int (default=200)*) – The size of buffer

model

Two best model for view1 and view2.

Type object list

fit(*X*, *targets*, *labeled_idx*)

Fit a cotraining model

All the input data is provided matrix X (labeled and unlabeled) and corresponding label matrix y with a dedicated marker value for unlabeled samples.

Parameters

- **feature1** (*view1 array-like, shape = [n_samples, n_features]*) –
- **feature2** (*view2 array_like, shape = [n_samples, n_features]*) –

- **targets** (*array_like, shape = [n_samples]*) – label of n_labeled_samples in X.
- **labeled_ind** (*the index of labeled data in targets.*) –
- **example** (*For*) –

Returns self

Return type returns an instance of self.

predict (*X*, *select_1=True*)

Parameters

- **X** (*np.ndarray, shape = [n_samples, n_features]*) – samples to be predicted
- **select_1** (*boolean, optional*) – select the prediction of model1 and model2.

Returns **y** – Predictions for input data

Return type np.ndarray, shape = [n_samples]

set_params (*param*)

Update the parameters of the estimator and release old results to prepare for new training.

LPA

class s3l.classification.LPA.**LPA** (*kernel='rbf'*, *gamma=20*, *n_neighbors=7*, *max_iter=30*, *tol=0.001*, *n_jobs=None*)

Bases: *s3l.base.TransductiveEstimatorwithGraph*

Class for label propagation module.

Parameters

- **kernel** (*{'knn', 'rbf', callable} (default='rbf')*) – String identifier for kernel function to use or the kernel function itself. Only 'rbf' and 'knn' strings are valid inputs. The function passed should take two inputs, each of shape [n_samples, n_features], and return a [n_samples, n_samples] shaped weight matrix.
- **gamma** (*float (default=20)*) – Parameter for rbf kernel
- **n_neighbors** (*integer > 0 (default=7)*) – Parameter for knn kernel
- **max_iter** (*integer (default=30)*) – Change maximum number of iterations allowed
- **tol** (*float (default=1e-3)*) – Convergence tolerance: threshold to consider the system at steady state
- **n_jobs** (*int or None, optional (default=None)*) – The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend`context. `-1'' means using all processors. See Glossary for more details.

fit (*X*, *y*, *labeled_idx*, *W*)

Fit a label propagation model

All the input data is provided matrix X (labeled and unlabeled) and corresponding label matrix y with a dedicated marker value for unlabeled samples. Optional matrix W is a graph provided for label propagation.

Parameters

- **X** (*array-like, shape = [n_samples, n_features]*) – A {n_samples by n_samples} size matrix will be created from this

- **y** (*array_like, shape = [n_samples]*) – n_labeled_samples (unlabeled points are marked as 0)
- **labeled_idx** (*array_like, shape = [n_samples]*) – index of n_labeled_samples in X.
- **W** (*array_like, shape = [n_samples, n_samples]*) – graph of instances

Returns self

Return type returns an instance of self.

predict(*index*)

Performs transductive inference across the model.

Parameters **index** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.

Returns y – Predictions for input data

Return type array_like, shape = [n_samples]

set_params(*param*)

Parameter setting function.

Parameters **paramdict** – Store parameter names and corresponding values {'name': value}.

TSVM

class s3l.classification.TSVM.**TSVM**(*kernel='RBF', C1=100, C2=0.1, alpha=0.1, beta=-1, gamma=0*)

Bases: *s3l.base.InductiveEstimatorWOGraph*

TSVM classifier

Parameters

- **kernel** (*{'Linear', 'RBF'} (default='RBF')*) – String identifier for kernel function to use or the kernel function itself. Only 'Linear' and 'RBF' strings are valid inputs.
- **C1** (*float (default=100)*) – Initial weight for labeled instances.
- **C2** (*float (default=0.1)*) – Initial weight for unlabeled instances.
- **alpha** (*float (default=0.1)*) – Balance parameter
- **beta** (*float (default=-1)*) – Balance parameter
- **gamma** (*float (default=0)*) – Parameter for RBF kernel

Other Parameters **model** (*object*) – Best model.

fit(*X*, *y*, *labeled_idx*)

Fit a semi-supervised SVM model

All the input data is provided matrix X (labeled and unlabeled) and corresponding label matrix y with a dedicated marker value for unlabeled samples.

Parameters

- **X** (*array-like, shape = [n_samples, n_features]*) – A {n_samples by n_samples} size matrix will be created from this
- **y** (*array_like, shape = [n_samples]*) – n_labeled_samples (unlabeled points are marked as 0)

- **labeled_idx** (*array_like, shape = [n_samples]*) – index of n_labeled_samples in X.

Returns self

Return type returns an instance of self.

predict(*X*)
Performs inductive inference across the model.

Parameters **X** (*array_like, shape = [n_samples, n_features]*) –

Returns **y** – Predictions for input data

Return type array_like, shape = [n_samples]

set_params(*param*)
Parameter setting function.

Parameters **paramdict** – Store parameter names and corresponding values {'name': value}.

data_quality

LEAD

This module implements the algorithm LEAD.

References

License: MIT

class s3l.data_quality.LEAD.**LEAD**(*C1=1.0, C2=0.01*)
Bases: *s3l.base.TransductiveEstimatorwithGraph*

Parameters

- **C1** (*float (default=1.0)*) – weight for the hinge loss of labeled instances. It was set as 1 in our paper.
- **C2** (*float (default=0.01)*) – weight for the hinge loss of unlabeled instances. It was set as 0.01 in our paper.

Examples

```
>>> from s3l.data_quality.LEAD import LEAD
>>> from s3l.datasets import data_manipulate, base
>>> X, y = base.load_covtype(True)
>>> W = base.load_graph_covtype(True)
>>> _, test_idxs, labeled_idxs, unlabeled_idxs = \
>>>                     data_manipulate.inductive_split(X=X, y=y)
>>> lead = LEAD(C1 = 1.0, C2 = 0.01)
>>> lead.fit(X,y,labeled_idxs,W)
>>> lead.predict(unlabeled_idxs)
[1,-1,-1,1,1...,1]
```

References

LEAD implements the LEAD algorithm in [1].

LEAD employs the Python version of liblinear [2] (available at http://www.csie.ntu.edu.tw/~cjlin/liblinear/).

fit (*gssl_value*, *label*, *l_ind*, *W*)
Given prediction from gssl, train method judge the quality of prediction with large-margin model

Parameters

- **gssl_value** (*array-like*) – a matrix with size n * T, where n is the number of instances and T is the number of graphs that gssl takes.Each row is a set of predictive values of an instance.
- **label** (*array-like*) – a column binary vector with length n. Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.
- **W** (*matrix*) – affinity matrix, labels should be at the left-top corner, should be in sparse form.

predict (*u_ind*, *baseline_pred=None*)
predict method replace the unsafe prediction with the baseline_pred to improve the safeness.

Parameters

- **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.
- **baseline_pred** (*array-like*) – a column vector with length n. Each element is a baseline predictive result of the corresponding instance. LEAD will replace the result of S3VM with this if the instance locates in the margin of S3VM.

Returns **pred** – the label of the instance, including labeled and unlabeled instances, even though for labeled instances the prediction is consistent with the true label.

Return type a column vector with length n. Each element is a prediction for

set_params (*param*)
Parameter setting function.

Parameters **dict** (*param*) – Store parameter names and corresponding values {'name': value}.

SLP

This module implements the algorithm SLP.

References

Author: De-Ming Liang <XXX@gmail.com> Xiao-Shuang Lv <XXX@XXX.com>

License: MIT

class s3l.data_quality.SLP.**SLP** (*stepSize=0.1*, *T=6*)
Bases: *s3l.base.TransductiveEstimatorwithGraph*

This is a python implementation of SLP, which can do label propagation on large-scale graphs.

Read more in the User Guide.

Parameters

- **stepSize** (*coefficient, optical (default=0.1)*) – step size.
- **T** (*coefficient,optical (default=6)*) – running epoches.

Examples

```
>>> from s3l.data_quality.SLP import SLP
>>> from s3l.datasets import data_manipulate, base
>>> X, y = base.load_covtype(True)
>>> W = base.load_graph_covtype(True)
>>> _, test_idxs, labeled_idxs, unlabeled_idxs = \
>>>                         data_manipulate.inductive_split(X=X, y=y)
>>> slp = SLP(stepSize=0.1, T=6)
>>> slp.fit(X,y,labeled_idxs,W)
>>> slp.predict(unlabeled_idxs)
[1,-1,-1,1,1...,1]
```

References

SLP implements the LEAD algorithm in [1].

fit (*X*, *y*, *l_ind*, *W*)
Fit the model to data.

Parameters

- **W** (*sparse matrix*) – affinity matrix, labels should be at the left-top corner, should be in sparse form.
- **y** (*array-like*) – label vector with different labels [n_samples].
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.

Returns **pred** – prediction of labels [n_samples, n_labels], in the original sort.

Return type array-like

predict (*u_ind*)
Compute the most possible label for samples in W.

Parameters **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.

Returns **pred** – Each row is the most likely label for a sample [n_samples].

Return type array-like

predict_proba (*u_ind*)
Compute probabilities of possible labels for samples in W.

Parameters **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.

Returns **pred** – Each line is the probability of possible labels of a sample involved in the calculation of the prediction [n_samples, n_labels].

Return type array-like

set_params (*param*)
Parameter setting function.

Parameters **dict** (*param*) – Store parameter names and corresponding values {'name': value}.

datasets

base

Base IO code for all datasets

s3l.datasets.base.**get_data_home**(*data_home=None*)
Return the path of the data dir.

This folder is used by some large dataset loaders to avoid downloading the data several times.

If the folder does not already exist, it is automatically created.

Parameters **data_home** (*str | None*) – The path to data dir.

s3l.datasets.base.**clear_data_home**(*data_home=None*)
Delete all the content of the data home cache.

Parameters **data_home** (*str | None*) – The path to data dir.

s3l.datasets.base.**load_data**(*feature_file=None*, *label_file=None*)
Load data from absolute path.

Parameters

- **feature_file** (*string.optional (default=None)*) – The absolute path of the user-provided feature dataset. The File should be in '.csv' format and organized as follows:

feature_name:	[1,n_features]
data:	[m_samples,n]

When the feature is a sparse matrix, the file should be in '*./mat/npz' format.

- **label_file** (*string.optional (default=None)*) – The absolute path of the user-provided label dataset. The File should be in '.csv' format and organized as follows:

label_name:	[1,n_labels]
label:	[m_samples,n]

When the label is a sparse matrix, the file should be in '*./mat/npz' format.

Besides,the number of rows in the label_file should be the same as the feature_file.

Returns

- **X** (*array-like*) – Data matrix with [m_samples, n_features].The data will be used to train models.
- **y** (*array-like*) – The label of load data with [m_samples, n_labels].

s3l.datasets.base.**load_graph**(*name*, *graph_file=None*)
Load graph from self-contained data set or user-provided data set. The self-contained data set is loaded first according to the provided data set name. Load the dataset according to the provided path when the dataset name is empty or does not exist.

Parameters

- **name** (*string.optional (default=None)*) – Name should be the name of the data in the self-contained data list.
- **graph_file** (*string.optional (default=None)*) – The absolute path of the user-provided feature dataset. The File should be in '*.csv/mat/npz' format .

Returns **W**

Return type np.nda

s3l.datasets.base.**load_boston**(*return_X_y=False*)
Load and return the boston house-prices dataset (regression).

Samples total	506
Dimensionality	13
Features	real, positive
Targets	real 5. - 50.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the regression targets, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if return_X_y is True)

s3l.datasets.base.**load_diabetes**(*return_X_y=False*)
Load and return the diabetes dataset (regression).

Samples total	442
Dimensionality	10
Features	real, -.2 < x < .2
Targets	integer 25 - 346

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn and 'target', the regression target for each sample.
- **(data, target)** (tuple if return_X_y is True)

s3l.datasets.base.**load_digits**(*return_X_y=False*)
Load and return the digits dataset (classification).

Each datapoint is a 8x8 image of a digit.

Classes	10
Samples per class	~180
Samples total	1797
Dimensionality	64
Features	integers 0-16

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'images', the images corresponding to each sample, 'target', the classification

labels for each sample, 'target_names', the meaning of the labels, and 'DESCR', the full description of the dataset.

- **(data, target)** (tuple if `return_X_y` is True)

This is a copy of the test set of the UCI ML hand-written digits datasets http://archive.ics.uci.edu/ml/datasets/Optical+Recognition+of+Handwritten+Digits

`s3l.datasets.base.`**`load_iris`**(*return_X_y=False*)
Load and return the iris dataset (classification).

The iris dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	50
Samples total	150
Dimensionality	4
Features	real, positive

Read more in the User Guide.

Parameters **`return_X_y`** (*`boolean, default=False.`*) – If True, returns `(data, target)` instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if `return_X_y` is True)

`s3l.datasets.base.`**`load_breast_cancer`**(*return_X_y=False*)
Load and return the breast cancer wisconsin dataset (classification).

The breast cancer dataset is a classic and very easy binary classification dataset.

Classes	2
Samples per class	212(M),357(B)
Samples total	569
Dimensionality	30
Features	real, positive

Parameters **`return_X_y`** (*`boolean, default=False`*) – If True, returns `(data, target)` instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if `return_X_y` is True)

The copy of UCI ML Breast Cancer Wisconsin (Diagnostic) dataset is downloaded from: https://goo.gl/U2Uwz2

`s3l.datasets.base.`**`load_linnerud`**(*return_X_y=False*)
Load and return the linnerud dataset (multivariate regression).

Samples total	20
Dimensionality	3 (for both data and target)
Features	integer
Targets	integer

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data' and 'targets', the two multivariate datasets, with 'data' corresponding to the exercise and 'targets' corresponding to the physiological measurements, as well as 'feature_names' and 'target_names'.
- **(data, target)** (tuple if return_X_y is True)

s3l.datasets.base.**load_wine**(*return_X_y=False*)
Load and return the wine dataset (classification).

The wine dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	[59,71,48]
Samples total	178
Dimensionality	13
Features	real, positive

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if return_X_y is True)

The copy of UCI ML Wine Data Set dataset is downloaded and modified to fit standard format from: https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data

s3l.datasets.base.**load_ionosphere**(*return_X_y=False*)
Load and return the ionosphere dataset (classification).

The ionosphere dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per class	[225,126]
Samples total	351
Dimensionality	34
Features	good, bad

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if return_X_y is True)

The copy of UCI ML ionosphere Data Set dataset is downloaded and modified to fit standard format from: https://archive.ics.uci.edu/ml/machine-learning-databases/ionosphere/ionosphere.data

s3l.datasets.base.**load_australian**(*return_X_y=False*)
Load and return the australian dataset (classification).

The australian dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per class	[307,383]
Samples total	690
Dimensionality	14
Features	class_1, class_0

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if return_X_y is True)

The copy of UCI ML australian Data Set dataset is downloaded and modified to fit standard format from: https://archive.ics.uci.edu/ml/machine-learning-databases/statlog/australian/australian.dat

s3l.datasets.base.**load_bupa**(*return_X_y=False*)
Load and return the bupa dataset (classification).

The bupa dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per class	[145,200]
Samples total	345
Dimensionality	6
Features	class_1, class_0

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=False.*) – If True, returns (data, target) instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if `return_X_y` is True)

The copy of UCI ML bupa Data Set dataset is downloaded and modified to fit standard format from: https://archive.ics.uci.edu/ml/machine-learning-databases/liver-disorders/bupa.data

`s3l.datasets.base.`**`load_haberman`**(*return_X_y=False*)
Load and return the haberman dataset (classification).

The haberman dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per class	[225,81]
Samples total	306
Dimensionality	3
Features	class_1, class_2

Read more in the User Guide.

Parameters **`return_X_y`** (*boolean, default=False.*) – If True, returns `(X, y)` instead of a Bunch object. See below for more information about the *X* and *y* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if `return_X_y` is True)

The copy of UCI ML haberman Data Set dataset is downloaded and modified to fit standard format from: https://archive.ics.uci.edu/ml/machine-learning-databases/haberman/haberman.data

`s3l.datasets.base.`**`load_vehicle`**(*return_X_y=False*)
Load and return the vehicle dataset (classification).

The vehicle dataset is a classic and very easy multi-class classification dataset.

Classes	4
Samples per	class[137,148,168,143]
Samples total	596
Dimensionality	18
Features	class_1, class_2

Read more in the User Guide.

Parameters **`return_X_y`** (*boolean, default=False.*) – If True, returns `(data, target)` instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if `return_X_y` is True)

The copy of libsvm vehicle Data Set dataset is downloaded and modified to fit standard format from: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass/vehicle.scale

Besides,We dropped the missing data.

s3l.datasets.base.**load_covtype**(*return_X_y=False*)
Load and return the covtype dataset (classification).

The covtype dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per	class[297711,283301]
Samples total	581012
Dimensionality	54
Features	class_1, class_-1

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=True.*) – If True, returns (data, target) . See below for more information about the *data* and *target* object.

Returns **(data, target)**

Return type tuple if return_X_y is True

s3l.datasets.base.**load_housing10**(*return_X_y=False*)
Load and return the housing10 dataset (classification).

The housing10 dataset is a classic and very easy multi-class classification dataset.

Classes	•
Samples per	•
Samples total	506
Dimensionality	13
Features	continue.

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=True.*) – If True, returns (data, target) . See below for more information about the *data* and *target* object.

Returns **(data ,target)**

Return type tuple if return_X_y is True

s3l.datasets.base.**load_spambase**(*return_X_y=False*)
Load and return the spambase dataset (classification).

The spambase dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per	class[1813,2788]
Samples total	4601
Dimensionality	57
Features	class_1, class_-1

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=True.*) – If True, returns (data, target) . See below for more information about the *data* and *target* object.

Returns (data, target)

Return type tuple if `return_X_y` is True

`s3l.datasets.base.`**load_house**(*return_X_y=False*)
Load and return the spambase dataset (classification).

The spambase dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per	class[108,124]
Samples total	232
Dimensionality	16
Features	class_1, class_-1

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=True.*) – If True, returns `(data, target)` . See below for more information about the *data* and *target* object.

Returns (data, target)

Return type tuple if `return_X_y` is True

`s3l.datasets.base.`**load_clean1**(*return_X_y=False*)
Load and return the house dataset (classification).

The spambase dataset is a classic and very easy multi-class classification dataset.

Classes	2
Samples per	class[207,269]
Samples total	476
Dimensionality	166
Features	class_1, class_-1

Read more in the User Guide.

Parameters **return_X_y** (*boolean, default=True.*) – If True, returns `(data, target)` . See below for more information about the *data* and *target* object.

Returns (data, target)

Return type tuple if `return_X_y` is True

`s3l.datasets.base.`**load_dataset**(*name=None*, *feature_file=None*, *label_file=None*)
Load data from self-contained data set or user-provided data set. The self-contained data set is loaded first according to the provided data set name. Load the dataset according to the provided path when the dataset name is empty or does not exist.

Parameters

- **name** (*string.optional (default=None)*) – Name should be the name of the data in the self-contained data list.
- **feature_file** (*string.optional (default=None)*) – The absolute path of the user-provided feature dataset. The File should be in '.csv' format and organized as follows:

feature_name:	[1,n_features]
data:	[m_samples,n]

- **label_file** (*string.optional (default=None)*) – The absolute path of the user-provided label dataset. The File should be in '.csv' format and organized as follows:

label_name:	[1,n_labels]
label:	[m_samples,n]

Besides,the number of rows in the label_file should be the same as the feature_file.

Returns

- **X** (*array-like*) – Data matrix with [m_samples, n_features].The data will be used to train models.
- **y** (*array-like*) – The label of load data with [m_samples, n_labels].

data_manipulate

This file implements some useful functions used to manipulate the data features or labels.

s3l.datasets.data_manipulate.**inductive_split**(*X=None*, *y=None*, *instance_indexes=None*, *test_ratio=0.3*, *initial_label_rate=0.05*, *split_count=10*, *all_class=True*, *save_file=False*, *saving_path=None*, *name=None*)

Provided one of X, y or instance_indexes to execute the inductive split.

Return the indexs for train/test data, and labled/unlabeled data in train ones for each split. If X, y are both provided, the lengths of them should be the same.

Parameters

- **X** (*array-like, optional*) – Data matrix with [n_instances, n_features]
- **y** (*array-like, optional*) – labels of given data [n_instances, n_labels] or [n_instances]
- **instance_indexes** (*list, optional (default=None)*) – List contains instances' names, used for image datasets, or provide index list instead of data matrix. Must provide one of [instance_names, X, y]
- **test_ratio** (*float, optional (default=0.3)*) – Ratio of test set
- **initial_label_rate** (*float, optional (default=0.05)*) – Ratio of initial label set e.g. Initial_labelset*(1-test_ratio)*n_instances
- **split_count** (*int, optional (default=10)*) – Random split data _split_count times
- **all_class** (*bool, optional (default=True)*) – Whether each split will contain at least one instance for each class. If False, a totally random split will be performed.
- **save_file** (*boolean, optional (default=False)*) –
- **saving_path** (*str, optional (default='.')*) – Giving None to disable saving.
- **name** (*str, optional (default=None)*) – Dataset name.

Returns

- **train_idx** (*list*) – index of training set, shape like [n_split_count, n_training_indexes]
- **test_idx** (*list*) – index of testing set, shape like [n_split_count, n_testing_indexes]
- **label_idx** (*list*) – index of labeling set, shape like [n_split_count, n_labeling_indexes]
- **unlabel_idx** (*list*) – index of unlabeling set, shape like [n_split_count, n_unlabeling_indexes]

s3l.datasets.data_manipulate.**ratio_split**(*X=None*, *y=None*, *instance_indexes=None*, *unlabel_ratio=0.3*, *split_count=10*, *all_class=True*, *save_file=False*, *saving_path=None*, *name=None*)

Split the data into labeled and unlabeled set with given ratio.

Provide one of X, y or instance_indexes to execute the transductive split. If X, y are both provided, the lengths of them should be the same. If X, instance_indexes are both provided, the instance_indexes is used for split.

Parameters

- **X** (*array-like, optional*) – Data matrix with [n_instances, n_features]
- **y** (*array-like, optional*) – labels of given data [n_instances, n_labels] or [n_instances]
- **instance_indexes** (*list, optional (default=None)*) – List contains instances' names, used for image datasets, or provide index list instead of data matrix. Must provide one of [instance_names, X, y]
- **unlabel_ratio** (*float, optional (default=0.3)*) – Ratio of test set
- **split_count** (*int, optional (default=10)*) – Random split data _split_count times
- **all_class** (*bool, optional (default=True)*) – Whether each split will contain at least one instance for each class. If False, a totally random split will be performed.
- **save_file** (*boolean, optional (default=False)*) –
- **saving_path** (*str, optional (default='.')*) – Giving None to disable saving.
- **name** (*str, optional (default=None)*) – Dataset name.

Returns

- **train_idxs** (*list*) – index of training set, shape like [n_split_count, n_training_indexes]
- **test_idxs** (*list*) – index of testing set, shape like [n_split_count, n_testing_indexes]

s3l.datasets.data_manipulate.**cv_split**(*X=None*, *y=None*, *instance_indexes=None*, *k=3*, *split_count=10*, *all_class=True*, *save_file=False*, *saving_path=None*, *name=None*)

Split the data into labeled and unlabeled set with given ratio.

Provide one of X, y or instance_indexes to execute the transductive split. Use instance_indexes firstly.

Note:

1. For multi-label task, set all_class = False.
2. For classification, the label must not be float type

Parameters

- **X** (*array-like, optional*) – Data matrix with [n_instances, n_features]
- **y** (*array-like, optional*) – labels of given data [n_instances, n_labels] or [n_instances]
- **instance_indexes** (*list, optional (default=None)*) – List provides index list instead of X. Must provide one of [instance_names, X, y]
- **k** (*int, optional (default=3)*) – Parameter for k-fold split. k should be small enough when we have few label data.

- **split_count** (*int, optional (default=10)*) – Random split data _split_count times
- **all_class** (*bool, optional (default=True)*) – Whether each split will contain at least one instance for each class. If False, a totally random split will be performed.
- **save_file** (*boolean, optional (default=False)*) – A flag indicates whether to save the splits.
- **saving_path** (*str, optional (default='.')*) – Giving None to disable saving.
- **name** (*str, optional (default=None)*) – Dataset name.

Returns

- **train_idx** (*list*) – index of training set, shape like [n_split_count, n_training_indexes]
- **test_idx** (*list*) – index of testing set, shape like [n_split_count, n_testing_indexes]

s3l.datasets.data_manipulate.**split_load**(*path*, *name*)
Load split from path.

Parameters

- **path** (*str*) – Absolute path to a dir which contains train_idx.txt, test_idx.txt, label_idx.txt, unlabel_idx.txt.
- **name** (*str*) – The name of dataset. The file is stored as 'XXX_train/test_idx.txt/npy'

Returns

- **train_idx** (*list*) – index of training set, shape like [n_split_count, n_training_samples]
- **test_idx** (*list*) – index of testing set, shape like [n_split_count, n_testing_samples]
- **label_idx** (*list*) – index of labeling set, shape like [n_split_count, n_labeling_samples]
- **unlabel_idx** (*list*) – index of unlabeling set, shape like [n_split_count, n_unlabeling_samples]

s3l.datasets.data_manipulate.**check_y**(*y*, *binary=True*)
Transform label vector to proba matrix. Use for binary and multi-class tasks.

Parameters

- **y** (*np.ndarray*) – Original label vector.
- **binary** (*boolean (default=True)*) – Indicate different tasks.

Returns

- **labels** (*1-D np.ndarray*) – A vector store the original labels. The labels are sorted as in *y_t*.
- **y_t** (*np.ndarray*) – When binary == True, y_t is 1-D vector with {1,-1}. When binary == False, y_t is a matrix in the shape n_samples, n_classes.

s3l.datasets.data_manipulate.**check_inputs**(*X*, *y*, *binary=True*)
Transform the input label vector to proba matrix; Encode the str feature.

Parameters

- **X** (*np.ndarray*) – Features
- **y** (*np.ndarray*) – Labels

s3l.datasets.data_manipulate.**modify_y**(*y*, *ind*, *n_labels*, *binary=True*)
This function is the reverse function of *check_y*, which transfer the prediction from inner results to the origin labels.

Parameters

- **y** (*np.ndarray*) – Prediction
- **ind** (*np.ndarray*) – Index
- **n_labels** (*1-D np.ndarray*) – A vector store the original labels. The labels are sorted as in *y*.

usps

usps dataset.

Handwritten recognition for digital acquisition. There are a total of 9298 handwritten digital images in the library(both 16*16 pixel grayscale values, the grayscale values have been normalized).And the dataset is divided into train and test.

The dataset page is available from LIBSVM

https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#usps

And the dataset with pretreatment can be found from

http://lamda.nju.edu.cn/liangdm/data/usps.tar.gz

s3l.datasets.usps.**fetch_usps**(*data_home=None*, *subset='train'*, *download_if_missing=True*, *return_X_y=False*)

Load the filenames and data from the usps dataset.

Parameters

- **data_home** (*optional, default: None*) – Specify a download and cache folder for the datasets.
- **subset** (*'train' or 'test', optional*) – Select the dataset to load: 'train' for the training set, 'test' for the test set.
- **download_if_missing** (*optional, default: True*) – If False, raise an IOError if the data is not locally available instead of trying to download the data from the source site.
- **return_X_y** (*optional,default: false*) – If True, returns `(data, target)` instead of a Bunch object. See below for more information about the *data* and *target* object.

Returns

- **data** (*Bunch*) – Dictionary-like object, the interesting attributes are: 'data', the data to learn, 'target', the classification labels, 'target_names', the meaning of the labels, 'feature_names', the meaning of the features, and 'DESCR', the full description of the dataset.
- **(data, target)** (tuple if `return_X_y` is True)

ensemble

SafetyForecast

class s3l.ensemble.SafetyForecast.**SafetyForecast**(*C1=1.0*, *C2=0.01*, *estimators=None*)

Bases: *s3l.base.SaferEnsemble*

Provide a safer prediction when given a set of training models or predict results. Judge the quality of prediction with large-margin model.

Parameters

- **C1** (*float (default=1.0)*) – weight for the hinge loss of labeled instances. It was set as 1 in our paper.
- **C2** (*float (default=0.01)*) – weight for the hinge loss of unlabeled instances. It was set as 0.01 in our paper.
- **pred_values** (*predict values*) –
- **fallback_ind** (*fallback indexs of unsafe prediction.*) –
- **estimators** (*list of estimators,optional (default=None)*) – When 'estimators' is none, it means user should provide predictive results. When 'estimators' is a list,each member of list is a tuple. Each tuple is an initialization of a estimator and a description of its parameters [(estimator,fit_params)].

Examples

When the 'estimators' parameter is initialized, the calling method is roughly as follows:

```
>>> from s3l.classification.TSVM import TSVM
>>> from s3l.model_uncertainty.S4VM import S4VM
>>> estimator_list = [(TSVM(),False), (S4VM(),True)]
>>> model = SafetyForecast(estimators=estimator_list)
>>> model.fit(x, y, l_ind)
>>> model.predict(u_ind)
```

It can be used like this when the 'estimators' is None:

```
>>> model = SafetyForecast()
>>> model.fit(prediction, y, l_ind)
>>> model.predict(u_ind, baseline_pred)
```

fit (*X*, *y*, *l_ind*)
Provide an interface that can pass in multiple learners or predictive results.

Parameters

- **X** (*array-like*) – Data matrix with [n_samples, n_features] or a set of prediction.
- **y** (*array-like*) – Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.

fit_estimators (*X*, *y*, *l_ind*, *u_ind*)
Provide a training interface that trains multiple models and give a safer prediction of these models.

Parameters

- **X** (*array-like*) – Data matrix with [n_samples, n_features].The data will be used to train models.
- **y** (*array-like*) – Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.
- **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.

fit_pred(*prediction*, *label*, *l_ind*, *u_ind*)
Predict a safer result from predictions, train method judge the quality of prediction with large-margin model

Parameters

- **prediction** (*array-like*) – A set of prediction.Each row is a set of predictive values of an instance.Each col is a prediction result.
- **label** (*array-like*) – Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.
- **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.

predict(*u_ind*, *baseline_pred=None*)
Predict method replace the unsafe prediction with the baseline_pred to improve the safeness.

Parameters

- **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.
- **baseline_pred** (*array-like*) – Each element is a baseline predictive result of the corresponding instance. LEAD will replace the result of S3VM with this if the instance locates in the margin of S3VM.

Returns **pred** – the label of the instance, including labeled and unlabeled instances, even though for labeled instances the prediction is consistent with the true label.

Return type a column vector with length n. Each element is a prediction for

predict_proba()
Compute probabilities of possible labels for samples in W.

Parameters **None** –

Returns **pred** – Each line is the probability of possible labels of a sample involved in the calculation of the prediction [n_samples, n_labels].

Return type array-like

set_params(*param*)
Parameter setting function.

Parameters **dict** (*param*) – Store parameter names and corresponding values {'name': value}.

metrics

performance

Pre-defined Performance

Implement classical methods

The metric method is called by performance_metric(ground-truth, prediction, param_dict) All metric method would pop the parameters in the param_dict first.

s3l.metrics.performance.**accuracy_score**(*y_true*, *y_pred*, *param_dict=None*)
Accuracy classification score.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) _labels.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Predicted _labels, as returned by a classifier.
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional␣
↪Sample
weights.
```

Returns score

Return type float

s3l.metrics.performance.**zero_one_loss**(*y_true*, *y_pred*, *param_dict=None*)
Zero-one classification loss.

If normalize is `True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int). The best performance is 0.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) labels.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Predicted labels, as returned by a classifier.
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
normalize : bool, optional (default=True)
    If ``False``, return the number of misclassifications.
    Otherwise, return the fraction of misclassifications.

sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns **loss** – If `normalize == True`, return the fraction of misclassifications (float), else it returns the number of misclassifications (int).

Return type float or int,

s3l.metrics.performance.**roc_auc_score**(*y_true*, *y_score*, *param_dict=None*)
Compute Area Under the Receiver Operating Characteristic Curve (ROC AUC) from prediction scores.

Parameters

- **y_true** (*array, shape = [n_samples] or [n_samples, n_classes]*) – True binary _labels or binary label indicators. True binary labels. If labels are not either {-1, 1} or {0, 1}, then pos_label should be explicitly given.
- **y_score** (*array, shape = [n_samples] or [n_samples, n_classes]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers). For binary y_true, y_score is supposed to be the score of the class with greater label.
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
pos_label : int or str, optional, default=None
    Label considered as positive and others are considered␣
↪negative.
```

(continues on next page)

(continued from previous page)

```
sample_weight : array-like of shape = [n_samples], optional,
default=None
    Sample weights.
```

Returns auc

Return type float

s3l.metrics.performance.**get_fps_tps_thresholds**(*y_true*, *y_score*, *param_dict=None*)

Calculate true and false positives per binary classification threshold.

Parameters

- **y_true** (*array, shape = [n_samples]*) – True targets of binary classification
- **y_score** (*array, shape = [n_samples]*) – Estimated probabilities or decision function
- **param_dict** (*dict*) –

  A dictory saving the parameters including::

  pos_label [int or str, default=None] The label of the positive class

Returns

- **fps** (*array, shape = [n_thresholds]*) – A count of false positives, at index i being the number of negative samples assigned a score >= thresholds[i]. The total number of negative samples is equal to fps[-1] (thus true negatives are given by fps[-1] - fps).
- **tps** (*array, shape = [n_thresholds <= len(np.unique(y_score))]*) – An increasing count of true positives, at index i being the number of positive samples assigned a score >= thresholds[i]. The total number of positive samples is equal to tps[-1] (thus false negatives are given by tps[-1] - tps).
- **thresholds** (*array, shape = [n_thresholds]*) – Decreasing score values.

s3l.metrics.performance.**f1_score**(*y_true*, *y_pred*, *labels=None*, *pos_label=1*, *average='binary'*, *sample_weight=None*)

Compute the F1 score, also known as balanced F-score or F-measure

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

```
F1 = 2 * (precision * recall) / (precision + recall)
```

In the multi-class and multi-label case, this is the average of the F1 score of each class with weighting depending on the `average` parameter.

Read more in the User Guide.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) target values.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Estimated targets as returned by a classifier.
- **labels** (*list, optional*) – The set of labels to include when `average != 'binary'`, and their order if `average is None`. Labels present in the data can be excluded, for example to calculate a multiclass average ignoring a majority negative class, while labels not present in the data will result in 0 components in a macro average. For multilabel targets, labels are column indices. By default, all labels in `y_true` and `y_pred` are used in sorted order.

Changed in version 0.17: parameter *labels* improved for multiclass problem.

- **pos_label** (*str or int, 1 by default*) – The class to report if `average='binary'` and the data is binary. If the data are multiclass or multilabel, this will be ignored; setting `labels=[pos_label]` and `average != 'binary'` will report scores for that label only.
- **average** (*string, [None, 'binary' (default), 'micro', 'macro', 'samples', 'weighted']*) – This parameter is required for multiclass/multilabel targets. If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

  'binary': Only report results for the class specified by `pos_label`. This is applicable only if targets (`y_{true,pred}`) are binary.

  'micro': Calculate metrics globally by counting the total true positives, false negatives and false positives.

  'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

  'weighted': Calculate metrics for each label, and find their average weighted by support (the number of true instances for each label). This alters 'macro' to account for label imbalance; it can result in an F-score that is not between precision and recall.

  'samples': Calculate metrics for each instance, and find their average (only meaningful for multilabel classification where this differs from *accuracy_score()*).

- **sample_weight** (*array-like of shape = [n_samples], optional*) – Sample weights.

Returns **f1_score** – F1 score of the positive class in binary classification or weighted average of the F1 scores of each class for the multiclass task.

Return type float or array of float, shape = [n_unique_labels]

See also:

`fbeta_score()`, `precision_recall_fscore_support()`, `jaccard_score()`, `multilabel_confusion_matrix()`

References

Examples

```
>>> from sklearn.metrics import f1_score
>>> y_true = [0, 1, 2, 0, 1, 2]
>>> y_pred = [0, 2, 1, 0, 0, 1]
>>> f1_score(y_true, y_pred, average='macro')  # doctest: +ELLIPSIS
0.26...
>>> f1_score(y_true, y_pred, average='micro')  # doctest: +ELLIPSIS
0.33...
>>> f1_score(y_true, y_pred, average='weighted')  # doctest: +ELLIPSIS
0.26...
>>> f1_score(y_true, y_pred, average=None)
array([0.8, 0. , 0. ])
```

Notes

When `true positive + false positive == 0` or `true positive + false negative == 0`, f-score returns 0 and raises `UndefinedMetricWarning`.

s3l.metrics.performance.**hamming_loss**(*y_true*, *y_pred*, *param_dict=None*)
Compute the average Hamming loss.

The Hamming loss is the fraction of labels that are incorrectly predicted.

Parameters

- **y_true** (*1d array-like, or label indicator array / sparse matrix*) – Ground truth (correct) labels.
- **y_pred** (*1d array-like, or label indicator array / sparse matrix*) – Predicted labels, as returned by a classifier.
- **labels** (*array, shape = [n_labels], optional (default=None)*) – Integer array of labels. If not provided, labels will be inferred from y_true and y_pred.

Returns **loss** – Return the average Hamming loss between element of `y_true` and `y_pred`.

Return type float or int,

s3l.metrics.performance.**one_error**(*y_true*, *y_pred*, *param_dict=None*)

Compute the one_error,similar to 0/1-loss.

Parameters

- **y_true** (*array, shape = [n_samples] or [n_samples, n_classes]*) – True binary labels or binary label indicators.
- **y_score** (*array, shape = [n_samples] or [n_samples, n_classes]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns **one_error**

Return type float

s3l.metrics.performance.**coverage_error**(*y_true*, *y_score*, *param_dict=None*)
Coverage error measure. Compute how far we need to go through the ranked scores to cover all true labels. The best value is equal to the average number of labels in `y_true` per sample.

Ties in `y_scores` are broken by giving maximal rank that would have been assigned to all tied values.

Parameters

- **y_true** (*array, shape = [n_samples, n_labels]*) – True binary labels in binary indicator format.
- **y_score** (*array, shape = [n_samples, n_labels]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns **coverage_error**

Return type float

s3l.metrics.performance.**label_ranking_loss**(*y_true*, *y_score*, *param_dict=None*)
Compute Ranking loss measure.

Compute the average number of label pairs that are incorrectly ordered given y_score weighted by the size of the label set and the number of labels not in the label set.

Parameters

- **y_true** (*array or sparse matrix, shape = [n_samples, n_labels]*) – True binary labels in binary indicator format.
- **y_score** (*array, shape = [n_samples, n_labels]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns loss

Return type float

References

s3l.metrics.performance.**label_ranking_average_precision_score**(*y_true*, *y_score*, *param_dict=None*)
Compute ranking-based average precision

Parameters

- **y_true** (*array or sparse matrix, shape = [n_samples, n_labels]*) – True binary labels in binary indicator format.
- **y_score** (*array, shape = [n_samples, n_labels]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns score

Return type float

s3l.metrics.performance.**micro_auc_score**(*y_true*, *y_score*, *param_dict=None*)
Compute the micro_auc_score.

Parameters

- **y_true** (*array, shape = [n_samples] or [n_samples, n_classes]*) – True binary labels or binary label indicators.
- **y_score** (*array, shape = [n_samples] or [n_samples, n_classes]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns micro_auc_score

Return type float

s3l.metrics.performance.**Average_precision_score**(*y_true*, *y_score*, *param_dict=None*)

Compute average precision (AP) from prediction scores

AP summarizes a precision-recall curve as the weighted mean of precisions achieved at each threshold, with the increase in recall from the previous threshold used as the weight:

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n$$

Parameters

- **y_true** (*array, shape = [n_samples] or [n_samples, n_classes]*) – True binary labels or binary label indicators.
- **y_score** (*array, shape = [n_samples] or [n_samples, n_classes]*) – Target scores, can either be probability estimates of the positive class, confidence values, or non-thresholded measure of decisions (as returned by "decision_function" on some classifiers).
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples], optional
    Sample weights.
```

Returns average_precision

Return type float

s3l.metrics.performance.**minus_mean_square_error**(*y_true*, *y_pred*, *param_dict=None*)

Minus mean square error

Parameters

- **y_true** (*1d array-like*) – Ground truth (correct) values.
- **y_pred** (*1d array-like*) – Predicted values, as returned by a regressor.
- **param_dict** (*dict*) – A dictory saving the parameters including:

```
sample_weight : array-like of shape = [n_samples],
optional Sample weights.
```

Returns score

Return type float

model_uncertainty

S4VM

S4VM implements the S4VM algorithm in [1].

S4VM employs the Python version of libsvm [2] (available at https://www.csie.ntu.edu.tw/~cjlin/libsvm/).

References

class s3l.model_uncertainty.S4VM.**S4VM**(*kernel='RBF'*, *C1=100*, *C2=0.1*, *sample_time=100*, *gamma=0*, *n_clusters=10*)

Bases: *s3l.base.InductiveEstimatorWOGraph*

Base class for S4VM module.

Parameters

- **kernel** (*'RBF' or 'Linear' (default='RBF')*) – String identifier for kernel function to use or the kernel function itself. Only 'RBF' and 'Linear' strings are valid inputs.
- **gamma** (*float*) – Parameter gamma is the width of RBF kernel. Default value is average distance between instances.
- **C1** (*double (default=100)*) – Weight for the hinge loss of labeled instance.
- **C2** (*double (default=0.1)*) – Weight for the hinge loss of unlabeled instance. If C2 is set as 0, our S4VM will degenerate to standard SVM.
- **sampleTime** (*integer (default=100)*) – The sampling times for each sampleTime.
- **n_clusters** (*integer (default=10)*) – The number of clusters to form as well as the number of centroids to generate for K-means.

fit(*X*, *y*, *labeled_idx*)

Fit the model according to the given training data.

Parameters

- **X** (*{array-like, sparse matrix}, shape = [n_samples, n_features]*) – Training vector containing labeled and unlabeled instances, where n_samples in the number of samples and n_features is the number of features. All unlabeled samples will be transductively assigned labels.
- **y** (*array-like, shape = [n_labeled_samples]*) – Target vector relative to labeled instances in X.
- **labeled_idx** (*array-like, shape = [n_labeled_samples]*) – Index of labeled instances in X.

Returns self

Return type object

predict(*u_ind*)

Predict method replace the unsafe prediction with the baseline_pred to improve the safeness.

Parameters **u_ind** (*array-like*) – a row vector with length l, where l is the number of unlabeled instance. Each element is an index of a unlabeled instance.

Returns **pred** – the label of the instance, including labeled and unlabeled instances, even though for labeled instances the prediction is consistent with the true label.

Return type a column vector with length n. Each element is a prediction for

set_params(*param*)

Parameter setting function.

Parameters **dict** (*param*) – Store parameter names and corresponding values {'name': value}.

SAFER

SAFER implements the SAFER algorithm in [1].

References

class s3l.model_uncertainty.SAFER.**SAFER**(*estimator=False*)

Bases: *s3l.base.InductiveEstimatorWOGraph*

baseline_predict(*X*, *y*, *l_ind*)

This is a 1NN regressor with euclidean distance measure.

Parameters

- **X** (*array-like*) – Data matrix with [n_samples, n_features] or a set of prediction.
- **y** (*array-like*) – Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.

fit(*X*, *y*, *l_ind=None*)

Provide an interface that can pass in multiple learners or predictive results.

Parameters

- **X** (*array-like*) – Data matrix with [n_samples, n_features] or a set of prediction.
- **y** (*array-like*) – Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like, optional (default=None)*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.

fit_estimator(*X*, *y*, *l_ind*, *n_neighbors=3*, *metric='minkowski'*)

Provide a training interface that trains multiple models and give a safer prediction of these models.

Parameters

- **X** (*array-like*) – Data matrix with [n_samples, n_features] or a set of prediction.
- **y** (*array-like*) – Each element is +1 or -1 for labeled instances. For unlabeled instances, this parameter could be used for computing accuracy if the ground truth is available.
- **l_ind** (*array-like*) – a row vector with length l, where l is the number of labeled instance. Each element is an index of a labeled instance.

fit_pred(*candidate_prediction=None*, *baseline_prediction=None*)

SAFER implements the SAFER algorithm in [1].

Parameters

- **candidate_prediction** (*array-like, optical (default=None)*) – a matrix with size instance_num * candidate_num . Each column vector of candidate_prediction is a candidate regression result.
- **baseline_prediction** (*array-like, optical (default=None)*) – a column vector with length instance_num. It is the regression result of the baseline method.

Returns **Safer_prediction** – a predictive regression result by SAFER.

Return type array-like

predict(*u_ind*)
Compute the most possible label for samples in X.

Returns **pred** – Each row is the most likely label for a sample [n_samples].

Return type array-like

set_params(*param*)
Parameter setting function.

Parameters **dict** (*param*) – Store parameter names and corresponding values {'name': value}.

utils

log_utils

s3l.utils.log_utils.**get_logger**(*name='s3l'*, *level=None*)
Fetch the logger with name and level.

Parameters

- **name** (*string, optional (default="s3l")*) –
- **level** (*const int, optional(default=None)*) – A value in the set {logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL} If None, the returned logger will have the default logging level set by *update_default_level*.

Returns **logger** – Call logger.[debug/info/warning/error/critical](msg, *arg, **kwargs) to add logging. Refer to https://docs.python.org/3/howto/logging.html.

Return type A logger object

s3l.utils.log_utils.**init_fh**()
Initialize the global file_handler for LOGGER.

s3l.utils.log_utils.**strftime**(*t=None*)

s3l.utils.log_utils.**update_default_level**(*default_level*)
Update the logging level for logger.

Parameters **default_level** (*const int*) – A value in the set {logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL}

s3l.utils.log_utils.**update_default_logging_dir**(*default_logging_dir*)
Update the dir to place the logging file.

Parameters **default_logging_dir** (*string*) –

1.3.2 Submodules

Experiments

Class to implement the process of semi-supervised learning experiments.

class s3l.Experiments.**SslExperimentsWithoutGraph**(*transductive=True*, *n_jobs=1*, *metri_param={}*, *all_class=True*)
Bases: *s3l.base.BaseExperiments*

Semi-supervised learning experiments without graph.

This class implements a common process of SSL experiments in both transductive and inductive settings. It optimize the hyper-parameters using grid-search policy which is paralleled using multi-processing.

Parameters

- **transductive** (*boolean, optional (default=True)*) – The experiment is transductive if True else inductive.
- **n_jobs** (*int, optional (default=1)*) – The nunmber of jobs to run the experiemnt.
- **all_class** (*boolean, optional (default=True)*) – Whether all split should have all classes.

performance_metric_name
The name of the metric.

Type string, optional (default='accuracy_score')

metri_param
A dict store the

Type dict, optional (default={})

datasets
A list of tuple which store the information of datasets to run.

Type list

configs
A list of tuple which store the information of algorithms to evaluate.

Type list

performance_metric
A callable object which is the evaluating method.

Type callable

metri_param
A dict which store the parameters for self.performance_metric.

Type dict

Notes

1. Multi-processing requests the estimator to be picklable. You may refer to *__getstate__* and *__setstate__* methods when your self-defined estimator has some problems with serialization.

experiments_on_datasets(*unlabel_ratio=0.8*, *test_ratio=0.3*, *number_init=5*)
The datasets are splits randomly or based on given splits. Get Label/Unlabel splits for each dataset in this funciton and conduct experiments on them. Results are stored for each dataset.

Parameters

- **unlabel_ratio** (*float*) – The ratio of test data for each dataset.
- **number_init** (*int*) – Different label initializations for each dataset.

Returns **results** – {dataset_name: {config_name:[scores]} }

Return type dict

class s3l.Experiments.**SslExperimentsWithGraph**(*n_jobs=1*)
Bases: *s3l.base.BaseExperiments*

Semi-supervised learning experiments with graph.

This class implements a common process of SSL experiments in both transductive and inductive settings for graph-based methods. It optimize the hyper-parameters using grid-search policy which is paralleled using multi-processing.

Parameters

- **transductive** (*boolean, optional (default=True)*) – The experiment is transductive if True else inductive.
- **n_jobs** (*int, optional (default=1)*) – The nunmber of jobs to run the experiemnt.
- **all_class** (*boolean, optional (default=True)*) – Whether all split should have all classes.

performance_metric_name
The name of the metric.

Type string, optional (default='accuracy_score')

metri_param
A dict store the

Type dict, optional (default={})

datasets
A list of tuple which store the information of datasets to run.

Type list

configs
A list of tuple which store the information of algorithms to evaluate.

Type list

performance_metric
A callable object which is the evaluating method.

Type callable

metri_param
A dict which store the parameters for self.performance_metric.

Type dict

Notes

1. Multi-processing requests the estimator to be picklable. You may refer to *__getstate__* and *__setstate__* methods when your self-defined estimator has some problems with serialization.

experiments_on_datasets(*unlabel_ratio=0.8*, *test_ratio=0.3*, *number_init=5*)
The datasets are splits randomly or based on given splits. Get Label/Unlabel splits for each dataset in this funciton and conduct experiments on them. Results are stored for each dataset.

Parameters

- **unlabel_ratio** (*float*) – The ratio of unlabeled data for each dataset.
- **test_ratio** (*float*) – The ratio of test data for each dataset. Is invalid when 'transductive'=True.
- **number_init** (*int*) – Different label initializations for each dataset.

Returns **results** – {dataset_name: {config_name:[scores]} }

Return type dict

base

Base classes for all estimators and experiments.

class s3l.base.**BaseEstimator**
Bases: abc.ABC

Base class for all estimators in s3l. .. rubric:: Notes

All estimators should specify all the parameters that can be set at the class level in their __init__ as explicit keyword arguments (no *args or **kwargs).

get_params(*deep=True*)
Get parameters for this estimator.

Parameters **deep** (*boolean, optional*) – If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns **params** – Parameter names mapped to their values.

Return type mapping of string to any

set_params(*param*)
Update the parameters of the estimator and release old results to prepare for new training.

class s3l.base.**BaseExperiments**(*transductive=True*, *n_jobs=1*, *all_class=True*)
Bases: object

The base class for all experiments. You can inherit this class to design you own experiment process.

append_configs(*configs*)
Append estimators configs to self.config

Parameters **configs** (*list of (name, estimator, param_dict)*) – In which name: string, estimator: object of estimator, param_dict: dict of parameters for corresponding estimator.

append_datasets(*datasets*)
Append datasets file names to self.datasets

Parameters **datasets** (*list of (name,feature_file,label_file, split_path,graph_file)*) – Detais:

```
name: string
    Name of the dataset. Arbitrary
feature_file: string or None
    Absolute file name of the feature file. Can be any thing if
label_file: string or None
    Absolute file name of the label file.
split_path: string or None
    Absolute path in which store the split files. Should be None␣
↪if
    no split files is provided.
graph_file: string or None
    Absolute file name of the graph files. Should be None if no
    graph is provided.
```

append_evaluate_metric(*performance_metric='accuracy_score'*, *kwargs={}*)
Append the metric for evaluation.

Parameters

- **performace_metric** (*str*) – The query performance-metric function. Giving str to use a pre-defined performance-metric.
- **kwargs** (*dict, optional*) – The args used in performance-metric. if kwargs is None,the pre-defined performance will init in the default way. Note that, each parameters should be static.

get_evaluation_results()

set_metric(*performance_metric='accuracy_score'*, *metric_large_better=True*, *param_dict=None*)
Set the metric for experiment.

Parameters

- **performace_metric** (*str*) – The query performance-metric function. Giving str to use a pre-defined performance-metric.
- **kwargs** (*dict, optional*) – The args used in performance-metric. if kwargs is None,the pre-defined performance will init in the default way. Note that, each parameters should be static.

class s3l.base.**InductiveEstimatorWOGraph**
Bases: *s3l.base.BaseEstimator*

fit(*X*, *y*, *l_ind*, ***kwargs*)
Takes X, y, label_index

predict(*X*, ***kwargs*)
Takes X

class s3l.base.**InductiveEstimatorwithGraph**
Bases: *s3l.base.BaseEstimator*

fit(*X*, *y*, *l_ind*, *W*, ***kwargs*)
Takes X, y, label_index, affinity matrix

predict(*X*, ***kwargs*)
Takes X

class s3l.base.**SaferEnsemble**
Bases: *s3l.base.BaseEstimator*

Base class for SaferEnsemble for semi-supervised learning. .. rubric:: Notes

All estimators should specify all the parameters that can be set at the class level in their `__init__` as explicit keyword arguments (no `*args` or `**kwargs`).

fit(*X*, *y*, *l_ind*, ***kwargs*)
Fit the model with base semi-supervised predictions.

predict(*u_ind*, *baseline_pred=None*)
Should provide baseline prediction. Can only make safer prediction with given ones, so it's transductive.

class s3l.base.**SupervisedEstimator**
Bases: *s3l.base.BaseEstimator*

Supervised estimator of single-label task.

fit(*X*, *y*, *l_ind=None*, ***kwargs*)
Takes X, y, label_index.

predict(*X*, ***kwargs*)
Takes X

predict_log_proba(*X*)

predict_proba(*X*)

set_params(*param*)
Update the parameters of the estimator and release old results to prepare for new training.

class s3l.base.**TransductiveEstimatorWOGraph**
Bases: *s3l.base.BaseEstimator*

fit (*X*, *y*, *l_ind*, ***kwargs*)
Takes X, y, label_index

predict (*u_ind*, ***kwargs*)
Takes unlabel_index

class s3l.base.**TransductiveEstimatorwithGraph**
Bases: *s3l.base.BaseEstimator*

fit (*X*, *y*, *l_ind*, *W*, ***kwargs*)
Takes X, y, label_index, affinity matrix

predict (*u_ind*, ***kwargs*)
Takes unlabel_index

1.4 User Guide

This section details the customization process and implementation of the built-in classes and povides further instructions for advanced users.

1.4.1 Usages of Built-in Experiments

In this page, we will introduce the details of the built-in experiments classes, which include classes of inductive/transductive semi-supervised learning with or without graph.

The Process of Built-in Experiment Class

We first introduce the main process of the experiment, which includes `load dataset`, `data manipulate`, `hyper-parameters selection` and `model evaluation`. Before the experiemnts start, you'll be asked to configure the datasets, evaluation metrics, estimators and their candidate parameters. Then, the experiments classes help you finish the whole process without much attention. You should set the estimators, metrics and the datasets to run the experiments.

We provide two built-in experiment classes: `SslExperimentsWithoutGraph`, *SslExperimentsWithGraph*. The common steps to run the experiment is:

- Initialize an instance of experiments class
- Set the estimators to evaluate with `append_configs` method
- Set the datasets with `append_datasets`
- Set the evaluation metrics with `set_metric` and `append_evaluate_metric` methods
- Run

Five Steps to Run the Experiments

Here is an example:

```
from s3l.Experiments import SslExperimentsWithoutGraph
from s3l.model_uncertainty.S4VM import S4VM

# list of (name, estimator instance, dict of parameters)
configs = [
        ('S4VM', S4VM(), {
            'kernel': 'RBF',
            'gamma':[0],
            'C1': [50,100],
```

(continues on next page)

(continued from previous page)

```
            'C2': [0.05,0.1],
            'sample_time':[100]
        })
    ]

# list of (name, feature_file, label_file, split_path, graph_file)
datasets = [
    ('house', None, None, None, None),
    ('isolet', None, None, None, None)
    ]

# 1. Initialize an object of experiments class
experiments = SslExperimentsWithoutGraph(transductive=True, n_jobs=4)
# 2. Set the estimators to evaluate with `append_configs` method
experiments.append_configs(configs)
# 3. Set the datasets with `append_datasets`
experiments.append_datasets(datasets)
# 4. Set the evaluation metrics with `set_metric`
experiments.set_metric(performance_metric='accuracy_score')
# optional. Additional metrics to evaluate the best model.
experiments.append_evaluate_metric(performance_metric='zero_one_loss')
experiments.append_evaluate_metric(performance_metric='hamming_loss')
# 5. Run
results = experiments.experiments_on_datasets(unlabel_ratio=0.75,test_ratio=0.2,
    number_init=2)
```

During the initialization, you should first choose a class from `SslExperimentsWithoutGraph`, `SslExperimentsWithGraph` based on using graph or not. Besides, you should specify the semi-supervised scheme as inductive (set `transductive``=``False`) or transductive (set `transductive``=``True`) and deside how many cores of CPU you want to use.

Then, you should call `append_configs`, `append_datasets` and `set_metric` in any order to configure the experiments:

- `append_configs` takes in a list of tuples like (name, object, parameters_dict). *name* is a string as you like, *object* is the object of an estimator, *parameters_dict* is a dict whose keys are name of parameters for corresponding estimator, values are lists of candidate values.
- `append_datasets` takes in a list of tuples like (name, feature_file, label_file, split_path, graph_file). *name* is a string used for output; *feature_file, label_file, split_spath, graph_file* can be string or NoneType, which should be the absolute path of the file you provided. If you use built-in datasets, *feature_file* and *label_file* can be None; If *split_file* is None, experiments class will split the data every time you run; *graph_file* should be set when the experiment need a graph.
- `set_metric` configures the evaluation metric used in `hyper-parameters selection`. The best model is selected based on this metric. [Here is a list of supported metrics](http). Please note that parameter `metric_large_better` indicates whether the metric is larger better.
- `append_evaluate_metric` appends other metrics which would be used to evaluate the best model selected in `hyper-parameters selection`. [Here is a list of supported metrics](http)

Attention

1. In order to reduce repetitive codes, we define some protocols to follow for estimators and metrics. You can refer to *How to Implement Your Own Estimators* for more details.
2. When debugging, you should close parallel mode by setting `n_jobs` to 1 else your code won't stop at the breakpoint.
3. If the built-in experiment process doesn't meet your demands, you can design yours own settings (refer to *How to Design Your Own Experiments* and *Experiments*).

1.4.2 Details of built-in Estimators

In this page, we will introduce the built-in estimator classes in each module. In short, they all has *fit*, *predict*, *set_params* methods. You can find more details in APIs to call them directly, or evaluate them in the *experiment* class.

About Classification

Three classical semi-supervised algorithms are implemented as the baselines of safe SSL, including Transductive Support Vector Machine (`TSVM`), Label Propagation Algorithm (`LPA`) and Co-training (`CoTraining`).

About Data Quality

Two algorithms called `LEAD` (LargE margin grAph quality juDgement) and `SLP` (Stochastic Label Propagation) are implemented in this package. `LEAD` is the first algorithm to study the quality of the graph. The basic idea is that given a set of candidate graphs, when one graph has a high quality, its predictive results may have a large margin separation. Therefore, given multiple graphs with unknown quality, one should encourage to use the graphs with a large margin, rather than the graphs with a small margin, and reduce the chances of performance degradation consequently. The proposed stacking method first regenerates a new SSL data set with the predictive results of GSSL on candidate graphs, and then formulates safe GSSL as the classical semi-supervised SVM optimization on the regenerated dataset.

`SLP` is a lightweight label propagation method for large-scale network data. A lightweight iterative process derived from the well-known stochastic gradient descent strategy is used to reduce memory overhead and accelerate the solving process.

About Model Uncertainty

We provide two safe SSL algorithms named `S4VM` (Safe Semi-supervised Support Vector Machine) and `SAFER` (SAFE semi-supervised Regression) in this package.

`S4VM` first generates a pool of diverse large margin low-density separators, and then optimizes the label assignment for the unlabeled data in the worse case under the assumption that the ground-truth label assignment can be realized by one of the obtained low-density separators.

For semi-supervised regression (SSR), `SAFER` tries to learn a safe prediction given a set of SSR predictions obtained in various ways. To achieve this goal, the safe semi-supervised regression problem is forumlated as a geometric projection issue. When the ground-truth label assignment is realized by a convex linear combination of base regressors, the proposal is probably safe and achieve the maximal worst-case performance gain.

About Ensemble

We also implement an ensemble method called `SafetyForest` to provide a safer prediction when given a set of training models or prediction results. `SafetyForest` works in a similar way as `LEAD`. The only difference between the two is that the input of the latter needs to be the predictions of graphs, but doest not need for the former.

About Wrapper

The wrapper helps wrapping the algorithms in third-party packages such as scikit-learn. We wrap some popular supervised learning algorithms in the wrapper as examples.

1.4.3 List of Built-in Metrics

This package provides some commonly used metrics. They are implemented in a pre-defined protocols in order to be called by experiments classes. Here is an example.

```
def accuracy_score(y_true, y_pred, param_dict=None):
    """Accuracy classification score.

    Parameters
    ----------
    y_true : 1d array-like, or label indicator array / sparse matrix
        Ground truth (correct) _labels.

    y_pred : 1d array-like, or label indicator array / sparse matrix
        Predicted _labels, as returned by a classifier.

    param_dict: dict
        A dictory saving the parameters including::

            sample_weight : array-like of shape = [n_samples], optional Sample
            weights.

    Returns
    -------
    score : float
    """
    # codes...
```

These functions always have three parameters: **y_true** for ground-truth labels, *y_pred* for prediction returned by an estimator, **param_dict** is a dict that stores other parameters used in the function.

The built-in metrics include:

```
'accuracy_score',
'zero_one_loss',
'roc_auc_score',
'get_fps_tps_thresholds',
'f1_score',
'hamming_loss',
'one_error',
'coverage_error',
'label_ranking_loss',
'label_ranking_average_precision_score',
'micro_auc_score',
'Average_precision_score',
'minus_mean_square_error'
```

Please refer to `s3l.metrics.performance` for more details.

1.4.4 How to Implement Your Own Estimators

All the estimators in all packages follow the pre-defined protocols based on their types. All the implementations of algorithms which follow the protocols in `s3l.base` can be evaluated as the built-in algorithms by experiment classes.

The estimators should inherit a base estimator class in `s3l.base` according to the type of the estimator you are going to implement. We currently provide five options for you:

1. `TransductiveEstimatorwithGraph`,
2. `TransductiveEstimatorWOGraph`,
3. `InductiveEstimatorWOGraph`,

4. `InductiveEstimatorwithGraph`,
5. `SupervisedEstimator`.

As the names indicate, the experiments support supervised learning algorithms, semi-supervised learning algorithms in both inductive and transductive settings with or without graph.

For each estimator class, you must implement the following methods: `set_params`, `fit` and `predict`.

`set_params` is the methods to configure the parameters of the estimator objects given a dict storing the values of some parameters. It's called in the experiments to search for the best hyper-parameters. Since the object is used repeatly with different hyper-parameters, **you should make sure that the object is reset as if hadn't been trained**. A common implementation is as follows.

```
def set_params(self, param):
        """Parameter setting function.

        Parameters
        ----------
        paramdict
            Store parameter names and corresponding values {'name': value}.
        """
        if isinstance(param, dict):
            self.__dict__.update(param)

        # Codes to reset some properties which may influence the
        # prediction.
```

`fit` is the method to train the model given data; `predict` is the method to make prediction. The main difference between base classes is the parameters of the `fit` and `predict`. For transductive estimator, the `predict` method takes in the indexes of instances to predict (the estimator can see the testing data when training). For inductive estimator, the `predict` method takes in the features of instances to predict. `fit` method always takes *X*, *y*, *l_ind*, and optional args are supported. For graph-based algorithms, *W* must be provided for `fit` method.

For supervised learning algorithm, you can inherit `SupervisedEstimator` class. You must rewrite `__init__` method and initialize the member *model* as an object of supervised learning model, and *model* must have the following methods:

```
class SupervisedEstimator(BaseEstimator):
    """ Supervised estimator of single-label task.
    """

    @abstractmethod
    def __init__(self):
        super(SupervisedEstimator, self).__init__()
        self.model = None

    def fit(self, X, y, l_ind=None, **kwargs):
        """
        Takes X, y, label_index.
        """
        if l_ind is not None:
            X = X[l_ind, :]
            if y.ndim == 2:
                y = y[l_ind, :].reshape(-1)
            else:
                y = y[l_ind]
        self.model.fit(X, y)

    def predict(self, X, **kwargs):
        """
        Takes X
        """
```

(continues on next page)

(continued from previous page)

```
        return self.model.predict(X)

    def set_params(self, param):
        self.model.set_params(**param)

    def predict_proba(self, X):
        return self.model.predict_proba(X)

    def predict_log_proba(self, X):
        return self.model.predict_log_proba(X)
```

`s3l.wrapper.sklearn_wrapper` guides you to wrap any supervised learning algorithm you like.

Attention

Sometimes your estimator class may contain *C-language* object member. The object of estimator can be un-serializable when the C object has pointers because the python interpreter has no way to know the details of the memory where the pointer points to.

The experiment classes run the experiemnts in multi-process mode when `n_jobs` is set larger than 1, which requires the estimator object is serializable. An option is to rewrite the `__getstate__` and `__setstate__` methods to design the way how estimator object is dumped and loaded by `pickle`. The simplest way is to drop the un-picklable member in `__getstate__` and re-initialze it in `__setstate__`. Here is an example taken from `s3l.classification.TSVM` where *self.model* is a C object:

```
def __getstate__(self):
    """
    The model is ctypes objects and contains pointers cannot be pickled.
    So we drop the model when we pickle TSVM.
    """
    state = self.__dict__.copy()
    del state['model']  # manually delete
    return state

def __setstate__(self, state):
    """
    The model is ctypes objects and contains pointers cannot be pickled.
    So we drop the model when we pickle TSVM.
    """
    self.__dict__.update(state)
    self.model = None  # manually update
```

1.4.5 Prepare Data

In this page, we will introduce the functions we provide to load datasets and split given data.

Load Data

In `s3l.datasets.base`, we provide some useful functions to load data. Here is the list:

```
'load_data',
'load_dataset',
'load_graph',
'load_boston',
'load_diabetes',
'load_digits',
'load_iris',
```

(continues on next page)

(continued from previous page)

```
'load_breast_cancer',
'load_linnerud',
'load_wine',
'load_ionosphere',
'load_australian',
'load_bupa',
'load_haberman',
'load_vehicle',
'load_covtype',
'load_housing10',
'load_spambase',
'load_house',
'load_clean1'
```

Among them, `load_data`, `load_dataset` and `load_graph` functions can be used to load the data you prepare. Other functions load the built-in datasets which are commonly used by researchers. These functions return the data in the form which can be used by estimators directly. For example,

```
X, y = load_XXX(return_X_y=False)
# XXX is the name of dataset
```

We'll show you how to use the two user-oriented functions `load_data`, `load_dataset` and `load_graph`. `load_dataset` is directly called in experiments classes, you can use them when you try algorithms outside experiment class or when you're implementing you own experiment class.

`load_data` loads features and labels of a dataset given the file names.

```
X, y = load_data(feature_file, label_file)
```

`load_dataset` wraps `load_data` with another parameter *name* and loads built-in dataset if *name* matchs.

```
X, y = load_dataset(name, feature_file, label_file)
```

`load_graph` loads the graph in `*.csv/npz/mat` file and returns a matrix.

```
W = load_graph(graph_file)
```

Split Data

In `s3l.datasets.data_manipulate`, we provide some useful functions to split data. Here is the list:

```
'inductive_split',
'ratio_split',
'cv_split'
```

Among them, `inductive_split` can split the dataset into three parts: labeled set, unlabeled set and testing set, which is helpful for semi-supervised learning tasks.

```
from sklearn.datasets import make_classification
from s3l.datasets import data_manipulate

X, y = make_classification()
train_idx, test_idx, label_idx, unlabel_idx = \
            data_manipulate.inductive_split(X, y,test_ratio=0.3,
                    initial_label_rate=0.05, split_count=10)
```

`ratio_split` and `cv_split` help split the given data based on train/test ratio and k-Fold.

```
from sklearn.datasets import make_classification
from s3l.datasets import data_manipulate

X, y = make_classification()
# ratio_split
train_idx, test_idx = \
            data_manipulate.ratio_split(X, y, unlabel_ratio=0.3,
                split_count=10)

# cv_split
train_idx, test_idx = \
            data_manipulate.cv_split(X, y, k=3, split_count=10)
```

The returned XXX_indexes are lists of indexes which can be directly used by built-in estimators.

1.5 Reference

This package is built based on the following works:

Works led by Dr. Li.

1. Yu-Feng Li, De-Ming Liang. Safe Semi-Supervised Learning: A Brief Introduction. Frontiers of Computer Science (FCS). in press.
2. De-Ming Liang, Yu-Feng Li. Lightweight label propagation for large-scale network data. In: Proceedings of the 27th International Joint Conference on Artificial Intelligence (IJCAI'18), Stockholm, Sweden, 2018, pp.3421-3427.
3. Yu-Feng Li, Han-Wen Zha, Zhi-Hua Zhou. Learning safe prediction for semi-supervised regression. In: Proceedings of the 31st AAAI conference on Artificial Intelligence (AAAI'17), San Francisco, CA, 2017, pp.2217-2223.
4. Yu-Feng Li, Shao-Bo Wang, Zhi-Hua Zhou. Graph quality judgement: A large margin expedition. In: Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI'16), New York, NY, 2016, pp.1725-1731.
5. Yu-Feng Li, James Kwok and Zhi-Hua Zhou. Towards safe semi-supervised learning for multivariate performance measures. In: Proceedings of the 30th AAAI conference on Artificial Intelligence (AAAI'16), Phoenix, AZ, 2016, pp. 1816-1822.
6. Yu-Feng Li and Zhi-Hua Zhou. Towards making unlabeled data never hurt. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI), 37(1):175-188, 2015.
7. Yu-Feng Li, Ivor Tsang, James Kwok and Zhi-Hua Zhou. Convex and Scalable Weakly Labeled SVMs. Journal of Machine Learning Research (JMLR), 14:2151-2188, 2013.
8. Yu-Feng Li and Zhi-Hua Zhou. Towards making unlabeled data never hurt. In: Proceedings of the 28th International Conference on Machine Learning (ICML'11), Bellevue, WA, 2011, pp.1081-1088.

Semi-supervised Learning

1. Blum, Avrim, and Tom Mitchell. Combining labeled and unlabeled data with co-training. In: Proceedings of the 11th Annual Conference on Computational Learning Theory (COLT'98), New York, NY, 1998, pp.92-100.
2. Joachims, Thorsten. Transductive Inference for Text Classi cation using Support Vector Machines. In: Proceedings of the 16th International Conference on Machine Learning (ICML'99), San Francisco, CA, 1999. pp.200-209.
3. Zhu, Xiaojin, Zoubin Ghahramani, and John D. Lafferty. Semi-Supervised Learning Using Gaussian Fields and Harmonic Functions. In: Proceedings of the 20th International conference on Machine learning (ICML'03), Washington, DC, 2003, pp. 912-919.

Third-party Library

1. Pedregosa, Fabian, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel et al. Scikit-learn: Machine learning in Python. Journal of machine learning research (JMLR), 12: 2825-2830, 2013.
2. C.-C. Chang and C.-J. Lin. LIBSVM : a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011..
3. R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification . Journal of Machine Learning Research (JMLR), 9: 1871-1874, 2008..

1.6 News

1.6.1 Plans and Release Note

v0.1.0

First release!

1. A general semi-supervised learning experiment framework.
2. Classical semi-supervised learning algorithms.
3. Some of our explorations on safe semi-supervised learning.

v0.1.1 (Developing)

1. Hope to improve generality to more settings.
2. Abstract the process of hyper-parameter optimization.
3. More algorithms.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

s

Index

G

H

I

L

M

O

P

R

S

T

U

Z