deslib Documentation

Release 0.3.5

Rafael M. O. Cruz

Feb 17, 2023

Contents

DESlib is an ensemble learning library focusing the implementation of the state-of-the-art techniques for dynamic classifier and ensemble selection.

DESlib is a work in progress. Contributions are welcomed through its GitHub page: https://github.com/scikit-learn-contrib/DESlib.

CHAPTER 1

Introduction

Dynamic Selection (DS) refers to techniques in which the base classifiers are selected on the fly, according to each new sample to be classified. Only the most competent, or an ensemble containing the most competent classifiers is selected to predict the label of a specific test sample. The rationale for such techniques is that not every classifier in the pool is an expert in classifying all unknown samples; rather, each base classifier is an expert in a different local region of the feature space.

DS is one of the most promising MCS approaches due to the fact that more and more works are reporting the superior performance of such techniques over static combination methods. Such techniques have achieved better classification performance especially when dealing with small-sized and imbalanced datasets. A comprehensive review of dynamic selection can be found in the following papers[1][2]

[1] : R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

[2] : A. S. Britto, R. Sabourin, L. E. S. de Oliveira, Dynamic selection of classifiers - A comprehensive review, Pattern Recognition 47 (11) (2014) 3665–3680.

CHAPTER 2

Philosophy

DESlib was developed with two objectives in mind: to make it easy to integrate Dynamic Selection algorithms to machine learning projects, and to facilitate research on this topic, by providing implementations of the main DES and DCS methods, as well as the commonly used baseline methods. Each algorithm implements the main methods in the scikit-learn API **scikit-learn**: **fit(X, y)**, **predict(X)**, **predict_proba(X)** and **score(X, y)**.

The implementation of the DS methods is modular, following a taxonomy defined in[1]. This taxonomy considers the main characteristics of DS methods, that are centered in three components:

1. the methodology used to define the local region, in which the competence level of the base classifiers are estimated (region of competence);
2. the source of information used to estimate the competence level of the base classifiers.
3. the selection approach to define the best classifier (for DCS) or the best set of classifiers (for DES).

This modular approach makes it easy for researchers to implement new DS methods, in many cases requiring only the implementation of the method **estimate_competence**, that is, how the local competence of the base classifier is measured.

CHAPTER 3

API Reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.1 User guide

This user guide explains how to install DESlib, how to contribute to the library and presents a step-by-step tutorial to fit and predict new instances using several dynamic selection techniques.

3.1.1 Installation

The library can be installed using pip:

Stable version:

```
pip install deslib
```

Latest version (under development):

```
pip install git+https://github.com/scikit-learn-contrib/DESlib
```

DESlib is tested to work with Python 3.5, 3.6 and 3.7. The dependency requirements are:

- scipy(>=0.13.3)
- numpy(>=1.10.4)
- scikit-learn(>=0.19.0)

These dependencies are automatically installed using the pip commands above.

Optional dependencies

To use Faiss (Fair AI Similarity Search), a fast implementation of KNN that can use GPUs, follow the instructions below: https://github.com/facebookresearch/faiss/blob/master/INSTALL.md

Note that Faiss is only available on Linux and MacOS.

3.1.2 Development

DESlib was started by Rafael M. O. Cruz as a way to facilitate research in this topic by providing other researchers a toolbox with everything that is required to easily develop and compare different dynamic ensemble techniques.

The library is a work in progress. As an open-source project, any type of contribution is welcomed and encouraged!

Contributing to DESlib

You can contribute to the project in several ways:

- Reporting bugs
- Requesting features
- Improving the documentation
- Adding examples to use the library
- Implementing new features and fixing bugs

Reporting Bugs and requesting features

We use Github issues to track all bugs and feature requests; feel free to open an issue if you have found a bug or wish to see a new feature implemented. Before opening a new issue, please check if the issue is not being currently addressed: [Issues](https://github.com/scikit-learn-contrib/DESlib/issues)

For reporting bugs:

- Include information of your working environment. This information can be found by running the following code snippet:

```
import platform; print(platform.platform())
import sys; print("Python", sys.version)
import numpy; print("NumPy", numpy.__version__)
import scipy; print("SciPy", scipy.__version__)
import sklearn; print("Scikit-Learn", sklearn.__version__)
```

- Include a [reproducible](https://stackoverflow.com/help/mcve) code snippet or link to a [gist](https://gist.github.com). If an exception is raised, please provide the traceback.

Documentation

We are glad to accept any sort of documentation: function docstrings, reStructuredText documents (like this one), tutorials, etc. reStructuredText documents live in the source code repository under the doc/ directory.

You can edit the documentation using any text editor and then generate the HTML output by typing `make html` from the doc/ directory. Alternatively, `make` can be used to quickly generate the documentation without the example gallery. The resulting HTML files will be placed in _build/html/ and are viewable in a web browser. See the README file in the doc/ directory for more information.

For building the documentation, you will need to install sphinx and sphinx_rtd_theme. This can be easily done by installing the requirements for development using the following command:

```
pip install -r requirements-dev.txt
```

Contributing with code

The preferred way to contribute is to fork the main repository to your account:

1. Fork the [project repository](https://github.com/scikit-learn-contrib/DESlib): click on the 'Fork' button near the top of the page. This creates a copy of the code under your account on the GitHub server.
2. Clone this copy to your local disk:

```
git clone git@github.com:YourLogin/DESlib.git
cd DESlib
```

3. Install all requirements for development:

```
pip install -r requirements-dev.txt
pip install --editable .
```

4. Create a branch to hold your changes:

```
git checkout -b branch_name
```

Where `branch_name` is the new feature or bug to be fixed. Do not work directly on the `master` branch.

5. Work on this copy on your computer using Git to do the version control. To record your changes in Git, then push them to GitHub with:

```
git push -u origin branch_name
```

It is important to assert your code is well covered by test routines (coverage of at least 90%), well documented and follows PEP8 guidelines.

6. Create a 'Pull request' to send your changes for review.

   If your pull request addresses an issue, please use the title to describe the issue and mention the issue number in the pull request description to ensure a link is created to the original issue.

3.1.3 Tutorial

This tutorial will walk you through generating a pool of classifiers and applying several dynamic selection techniques for the classification of unknown samples. The tutorial assumes that you are already familiar with the Python language and the scikit-learn library. Users not familiar with either Python and scikit-learn can start by checking out their tutorials.

Running Dynamic selection with Bagging

In this first tutorial, we do a step-by-step run of the example_bagging.py, that is included in the examples part of the DESlib. This example uses the Wisconsin breast cancer dataset available on sklearn.datasets package.

The first step is to run the example to check if everything is working as intended:

```
cd examples
python example_bagging.py
```

This script run six different dynamic selection models: Three DCS (OLA, A-Priori, MCB) and four DES (KNORA-Union, KNORA-Eliminate, DES-P and META-DES)

The example outputs the classification accuracy of each dataset:

```
Evaluating DS techniques:
Classification accuracy KNORA-Union:  0.973404255319
Classification accuracy KNORA-Eliminate:  0.968085106383
Classification accuracy DESP:  0.973404255319
Classification accuracy OLA:  0.968085106383
Classification accuracy A priori:  0.973404255319
Classification accuracy MCB:  0.968085106383
Classification accuracy META-DES:  0.973404255319
```

Code analysis:

The code starts by importing the corresponding DCS and DES algorithms that are tested as well as the other required libraries:

```
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import Perceptron
from sklearn.calibration import CalibratedClassifierCV
from sklearn.ensemble import BaggingClassifier

#importing DCS techniques from DESlib
from deslib.dcs.ola import OLA
from deslib.dcs.a_priori import APriori
from deslib.dcs.mcb import MCB

#import DES techniques from DESlib
from deslib.des.des_p import DESP
from deslib.des.knora_u import KNORAU
from deslib.des.knora_e import KNORAE
from deslib.des.meta_des import METADES
```

As DESlib is built on top of scikit-learn, code will usually required the import of routines from it.

Preparing the dataset:

The next step is loading the data and dividing it into three partitions: Training, validation and test. In the dynamic selection literature[1] the validation set is usually called the dynamic selection dataset (DSEL), since this partition is used by the dynamic selection techniques in order to select the base classifiers, so in this library we use the same terminology. The training set (X_train, y_train) is used to fit the pool of classifiers, the validation (X_DSEL, y_DSEL) set is used to fit the dynamic selection models. The performance of the system is then evaluated on the test set (X_test, y_test).

[1] : R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

```
data = load_breast_cancer()
X = data.data
y = data.target
# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)

# Scale the variables to have 0 mean and unit variance
scalar = StandardScaler()
X_train = scalar.fit_transform(X_train)
X_test = scalar.transform(X_test)

# Split the data into training and DSEL for DS techniques
X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train, test_size=0.5)
```

Another important aspect is to normalize the data so that it has zero mean and unit variance, which is a common requirement for many machine learning algorithms. This step can be easily done using the StandardScaler class from scikit-learn. Note that the StandardScaler transform should be fitted using the training and DSEL data only. Then, it can be applied for the test data.

An important point here is that in case of small datasets or when the base classifier models in the pool are weak estimators such as Decision Stumps or Perceptrons, an overlap between the training data and DSEL may be beneficial for achieving better performance.

Training a pool of classifiers:

The next step is to generate a pool of classifiers. This list can be either homogeneous (i.e., all base classifiers are of the same type) or heterogeneous (base classifiers of different types). The library supports any type of base classifiers that is compatible with the scikit-learn library.

In this example, we generate a pool composed of 10 Perceptron classifiers using the Bagging technique. It is important to mention that some DS techniques require that the base classifiers are capable of estimating probabilities (i.e., implements the predict_proba function).

For the Perceptron model, this can be achieved by calibrating the outputs of the base classifiers using the CalibratedClassifierCV class from scikit-learn.

```
model = CalibratedClassifierCV(Perceptron(max_iter=10))

# Train a pool of 10 classifiers
pool_classifiers = BaggingClassifier(model, n_estimators=10)
pool_classifiers.fit(X_train, y_train)
```

Building the DS models

Three DCS and four DES techniques are considered in this example:

- Overal Local Accuracy (OLA)
- Multiple-Classifier Behavior (MCB)
- A Priori selection
- K-Nearest Oracles-Union (KNU)
- K-Nearest Oracles-Eliminate (KNE)
- META-DES

NEW: Since version 0.3, DESlib does not require a trained pool of classifiers for instantiating its estimators. All estimator can now be instantiated without specifying a pool of classifiers:

```
# DCS techniques
ola = OLA()
mcb = MCB()
apriori = APriori()

# DES techniques
knorau = KNORAU()
kne = KNORAE()
desp = DESP()
meta = METADES()
```

When the pool of classifiers is not specified, a standard `BaggingClassifier` from sklearn is used, which generates a pool composed of 10 decision trees. The parameter **DSEL_perc** controls the percentage of the input data that is used for fitting DSEL. The remaining data will be used to fit the pool of classifiers. Note that this parameter is only taken into account if the pool is either equals to None (when it was not informed) or still unfitted (when the base classifiers were not fitted)

However, since we already trained a pool of classifiers in the previous step we will continue this tutorial by instantiating the dynamic selection methods with an already fitted pool. For more information on using DESlib estimators without specifying a trained pool of classifiers see the examples page.

```
# DCS techniques
ola = OLA(pool_classifiers)
mcb = MCB(pool_classifiers)
apriori = APriori(pool_classifiers)

# DES techniques
knorau = KNORAU(pool_classifiers)
kne = KNORAE(pool_classifiers)
desp = DESP(pool_classifiers)
meta = METADES(pool_classifiers)
```

Fitting the DS techniques:

The next step is to fit the DS model. We call the function fit to prepare the DS techniques for the classification of new data by pre-processing the information required to apply the DS techniques, such as, fitting the algorithm used to estimate the region of competence (k-NN, k-Means) and calculating the source of competence of the base classifiers for each sample in the dynamic selection dataset.

```
knorau.fit(X_dsel, y_dsel)
kne.fit(X_dsel, y_dsel)
desp.fit(X_dsel, y_dsel)
ola.fit(X_dsel, y_dsel)
mcb.fit(X_dsel, y_dsel)
apriori.fit(X_dsel, y_dsel)
meta.fit(X_dsel, y_dsel)
```

Note that if the pool of classifiers is still unfitted, this step will also fit the base classifiers in the pool.

Estimating classification accuracy:

Estimating the classification accuracy of each method is very easy. Each DS technique implements the function score from scikit-learn in order to estimate the classification accuracy.

```
print('Classification accuracy OLA: ', ola.score(X_test, y_test))
print('Classification accuracy A priori: ', apriori.score(X_test, y_test))
print('Classification accuracy KNORA-Union: ', knorau.score(X_test, y_test))
print('Classification accuracy KNORA-Eliminate: ', kne.score(X_test, y_test))
print('Classification accuracy DESP: ', desp.score(X_test, y_test))
print('Classification accuracy META-DES: ', apriori.score(X_test, y_test))
```

However, you may need to calculate the predictions of the model or the estimation of probabilities instead of only computing the accuracy. Class labels and posterior probabilities can be easily calculated using the **predict** and **predict_proba** methods:

```
metades.predict(X_test)
metades.predict_proba(X_test)
```

Changing parameters

Changing the hyper-parameters is very easy. We just need to pass its value when instantiating a new method. For example, we can change the size of the neighborhood used to estimate the competence level by setting the k value.

```
# DES techniques
knorau = KNORAU(pool_classifiers, k=5)
kne = KNORAE(pool_classifiers, k=5)
```

Also, we can change the mode DES algorithm works (dynamic selection, dynamic weighting or hybrid) by setting its mode: .. code-block:: python

    meta = METADES(pool_classifiers, Hc=0.8, k=5, mode='hybrid')

In this code block, we change the size of the neighborhood (k=5), the value of the sample selection mechanism (Hc=0.8) and also, state that the META-DES algorithm should work in a hybrid dynamic selection with and weighting mode. The library accepts the change of several hyper-parameters. A list containing each one for all technique available as well as its impact in the algorithm is presented in the API Reference.

Applying the Dynamic Frienemy Pruning (DFP)

The library also implements the Dynamic Frienemy Pruning (DFP) proposed in[2]. So any dynamic selection technique can be applied using the FIRE (Frienemy Indecision Region Dynamic Ensemble Selection) framework. That is easily done by setting the DFP to true when initializing a DS technique. In this example, we show how to start the FIRE-KNORA-U, FIRE-KNORA-E and FIRE-MCB techniques.

```
fire_knorau = KNORAU(pool_classifiers, DFP=True)
fire_kne = KNORAE(pool_classifiers, DFP=True)
fire_mcb = MCB(pool_classifiers, DFP=True)
```

We can also set the size of the neighborhood that is used to decide whether the query sample is located in a safe region or in an indecision region (safe_k):

[2] : Oliveira, D.V.R., Cavalcanti, G.D.C. and Sabourin, R., Online Pruning of Base Classifiers for Dynamic Ensemble Selection, Pattern Recognition, vol. 72, December 2017, pp 44-58.

```
fire_knorau = KNORAU(pool_classifiers, DFP=True, safe_k=3)
fire_kne = KNORAE(pool_classifiers, DFP=True, safe_k=5)
fire_mcb = MCB(pool_classifiers, DFP=True, safe_k=7)
```

So, the fire_knorau will use a neighborhood composed of 3 samples, fire_knorae of 5 and fire_mcb of 7 in order to compute whether a given sample is located in a indecision or safe region.

More tutorials on how to use different aspects of the library can be found in examples page

References

3.1.4 Known Issues

The estimators in this library are not compatible with scikit-learn's GridSearch, and other CV methods. That is, the following is not supported:

```
from deslib.des.knora_e import KNORAE
from sklearn.model_selection import GridSearchCV

# (...) initialize a pool of classifiers
kne = KNORAE(pool_classifiers)

# Do a grid search on KNORAE's "k" parameter
params = {'k': [1, 3, 5, 7]}

grid = GridSearchCV(kne, params)
grid.fit(X_dsel, y_dsel)  # Raises an error
```

This is due to a limitation of a scikit-learn method (sklearn.base.clone), under discussion in this issue

3.1.5 Releasing a new version

Publishing new version involves:

1) Updating the version numbers and creating a new tag in git (which also updates the "stable" version of the documentation)
2) Creating the distribution (.tar.gz and wheel files), and uploading them to pypi

Some important things to have in mind:

- Read the "Packaging and Distributing Projects" guide: https://packaging.python.org/tutorials/distributing-packages/
- The version numbers (in setup.py and __init__.py) are used as metadata for pypi and for the readthedocs documentation - pay attention to them or some things can break. In general, you should be working on a version such as "0.2.dev". You then rename it to "0.2" and create a tag "v0.2". After you finish everything, you update the version to "0.3.dev" to indicate that new developments are being made for the next version.

Step-by-step process

- Create an account in PyPi production: https://pypi.org/ and test: https://test.pypi.org/
- Make sure you have twine installed:

```
pip install twine
```

- Update version on setup.py (e.g. "0.1")
- Update version on deslib/__init__.py
- Create tag: `git tag <version>` (example: "git tag 'v0.1'")
- Push the tag `git push origin <version>`
- Create the source and wheels distributions

```
python setup.py sdist # source distribution
python setup.py bdist_wheel # wheel distribution for current python version
```

- Upload to test pypi and check
  - uploading the package:

```
twine upload --repository-url https://test.pypi.org/legacy/ dist/*
```

  - Note: if you do this multiple times (e.g. to fix an issue), you will need to rename the files under the "dist" folder: a filename can only be submitted once to pypi. You may also need to manually delete the "source" version of the distribution, since there can only be one source file per version of the software
  - Test an installation from the testing pypi environment.

```
conda create -y -n testdes python=3
source activate testdes
pip install --index-url https://test.pypi.org/simple/ --extra-index-url https://
↪pypi.org/simple deslib
conda remove -y --name testdes --all  #remove temporary environment
```

- Upload to production pypi

```
twine upload dist/*
```

- Mark the new stable version to be built on readthedocs:
- Go to https://readthedocs.org/projects/deslib/versions/, find the new tag and click "Edit". Mark the "active" checkbox and save.
- Update version on setup.py and __init.py__ to mention the new version in development (e.g. "0.2.dev")

Note #1: Read the docs is automatically updated:

- When a new commit is done in master (this updates the "master" version)
- When a new tag is pushed to github (this updates the "stable" version) -> This seems to not aways work - it is better to check

Note #2: The documentation automatically links to source files for the methods/classes. This only works if the tag is pushed to github, and matches the __version__ variable in __init.py__. Example: __version__ = "0.1" and the tag being: git tag "v0.1"

3.2 API Reference

This is the full API documentation of the *DESlib*. Currently the library is divided into four modules:

3.2.1 Dynamic Classifier Selection (DCS)

This module contains the implementation of techniques in which only the base classifier that attained the highest competence level is selected for the classification of the query.

The *deslib.dcs* provides a set of key dynamic classifier selection algorithms (DCS).

A posteriori

class deslib.dcs.a_posteriori.**APosteriori**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *selection_method='diff'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

A Posteriori Dynamic classifier selection.

The A Posteriori method uses the probability of correct classification of a given base classifier c_i for each neighbor x_k with respect to a single class. Consider a classifier c_i that assigns a test sample to class w_l. Then, only the samples belonging to class w_l are taken into account during the competence level estimates. Base classifiers with a higher probability of correct classification have a higher competence level. Moreover, the method also weights the influence of each neighbor x_k according to its Euclidean distance to the query sample. The closest neighbors have a higher influence on the competence level estimate. In cases where no sample in the region of competence belongs to the predicted class, w_l, the competence level estimate of the base classifier is equal to zero.

A single classifier is selected only if its competence level is significantly higher than that of the other base classifiers in the pool (higher than a pre-defined threshold). Otherwise, all classifiers in the pool are combined using the majority voting rule. The selection methodology can be modified by modifying the hyper-parameter selection_method.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict" and "predict_proba". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

G. Giacinto and F. Roli, Methods for Dynamic Classifier Selection 10th Int. Conf. on Image Anal. and Proc., Venice, Italy (1999), 659-664.

Ko, Albert HR, Robert Sabourin, and Alceu Souza Britto Jr. "From dynamic classifier selection to dynamic ensemble selection." Pattern Recognition 41.5 (2008): 1718-1731.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

Estimate the competence of each base classifier c_i for the classification of the query sample using the A Posteriori method.

The competence level is estimated based on the probability of correct classification of the base classifier c_i, for each neighbor x_k belonging to a specific class w_l. In this case, w_l is the class predicted by the base classifier c_i, for the query sample. This method also weights the influence of each training sample according to its Euclidean distance to the query instance. The closest samples have a higher influence in the computation of the competence level. The competence level estimate is represented by the following equation:

$$\delta_{i,j} = \frac{\sum_{\mathbf{x}_k \in \omega_l} P(\omega_l \mid \mathbf{x}_k, c_i) W_k}{\sum_{k=1}^{K} P(\omega_l \mid \mathbf{x}_k, c_i) W_k}$$

where $\delta_{i,j}$ represents the competence level of c_i for the classification of query.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS method.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

A Priori

class deslib.dcs.a_priori.**APriori**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *selection_method='diff'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

A Priori dynamic classifier selection.

The A Priori method uses the probability of correct classification of a given base classifier c_i for each neighbor x_k for the competence level estimation. Base classifiers with a higher probability of correct classification have a higher competence level. Moreover, the method also weights the influence of each neighbor x_k according to its Euclidean distance to the query sample. The closest neighbors have a higher influence on the competence level estimate.

A single classifier is selected only if its competence level is significantly higher than that of the other base classifiers in the pool (higher than a pre-defined threshold). Otherwise, all classifiers i the pool are combined using the majority voting rule.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict" and "predict_proba". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

G. Giacinto and F. Roli, Methods for Dynamic Classifier Selection 10th Int. Conf. on Image Anal. and Proc., Venice, Italy (1999), 659-664.

Ko, Albert HR, Robert Sabourin, and Alceu Souza Britto Jr. "From dynamic classifier selection to dynamic ensemble selection." Pattern Recognition 41.5 (2008): 1718-1731.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i for the classification of the query sample using the A Priori rule:

The competence level is estimated based on the probability of correct classification of the base classifier c_i, considering all samples in the region of competence. This method also weights the influence of each training sample according to its Euclidean distance to the query instance. The closest samples have a higher influence in the computation of the competence level. The competence level estimate is represented by the following equation:

$$\delta_{i,j} = \frac{\sum_{k=1}^{K} P(\omega_l \mid \mathbf{x}_k \in \omega_l, c_i) W_k}{\sum_{k=1}^{K} W_k}$$

where $\delta_{i,j}$ represents the competence level of c_i for the classification of query.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS method.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

Local Class Accuracy (LCA)

class deslib.dcs.lca.**LCA**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *selection_method='best'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *knne=False*, *n_jobs=-1*)

Local Class Accuracy (LCA).

Evaluates the competence level of each individual classifiers and select the most competent one to predict the label of each test sample. The competence of each base classifier is calculated based on its local accuracy with respect to some output class. Consider a classifier c_i that assigns a test sample to class w_l. The competence level of c_i is estimated by the percentage of the local training samples assigned to class w_l that it predicts the correct class label.

The LCA method selects the base classifier presenting the highest competence level. In a case where more than one base classifier achieves the same competence level, the one that was evaluated first is selected. The selection methodology can be modified by changing the hyper-parameter selection_method.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' : will use `KNeighborsClassifier` from sklearn

  `KNNE`.

- 'faiss' : will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- *None* : will use sklearn `KNeighborsClassifier`.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Woods, Kevin, W. Philip Kegelmeyer, and Kevin Bowyer. "Combination of multiple classifiers using local accuracy estimates." IEEE transactions on pattern analysis and machine intelligence 19.4 (1997): 405-410.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence of each base classifier c_i for the classification of the query sample using the local class accuracy method.

In this algorithm the k-Nearest Neighbors of the test sample are estimated. Then, the local accuracy of the base classifiers is estimated by its classification accuracy taking into account only the samples from the class w_l in this neighborhood. In this case, w_l is the class predicted by the base classifier c_i, for the query sample. The competence level estimate is represented by the following equation:

$$\delta_{i,j} = \frac{\sum_{\mathbf{x}_k \in \omega_l} P(\omega_l \mid \mathbf{x}_k, c_i)}{\sum_{k=1}^{K} P(\omega_l \mid \mathbf{x}_k, c_i)}$$

where $\delta_{i,j}$ represents the competence level of c_i for the classification of query.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

Multiple Classifier Behaviour (MCB)

class deslib.dcs.mcb.**MCB**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *similarity_threshold=0.7*, *selection_method='diff'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Multiple Classifier Behaviour (MCB).

The MCB method evaluates the competence level of each individual classifiers taking into account the local accuracy of the base classifier in the region of competence. The region of competence is defined using the k-NN and behavioral knowledge space (BKS) method. First the k-nearest neighbors of the test sample are computed. Then, the set containing the k-nearest neighbors is filtered based on the similarity of the query sample and its neighbors using the decision space (BKS representation).

A single classifier c_i is selected only if its competence level is significantly higher than that of the other base classifiers in the pool (higher than a pre-defined threshold). Otherwise, all classifiers in the pool are combined using the majority voting rule. The selection methodology can be modified by changing the hyper-parameter selection_method.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`

- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Giacinto, Giorgio, and Fabio Roli. "Dynamic classifier selection based on multiple classifier behaviour." Pattern Recognition 34.9 (2001): 1879-1881.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

Huang, Yea S., and Ching Y. Suen. "A method of combining multiple experts for the recognition of unconstrained handwritten numerals." IEEE Transactions on Pattern Analysis and Machine Intelligence 17.1 (1995): 90-94.

Huang, Yea S., and Ching Y. Suen. "The behavior-knowledge space method for combination of multiple classifiers." IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 1993.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence of each base classifier c_i for the classification of the query sample using the Multiple Classifier Behaviour criterion.

The region of competence in this method is estimated taking into account the feature space and the decision space (using the behaviour knowledge space method [4]). First, the k-Nearest Neighbors of the query sample are defined in the feature space to compose the region of competence. Then, the similarity in the BKS space between the query and the instances in the region of competence are estimated using the following equations:

$$S(\tilde{\mathbf{x}}_j, \tilde{\mathbf{x}}_k) = \frac{1}{M} \sum_{i=1}^{M} T(\mathbf{x}_j, \mathbf{x}_k)$$

$$T(\mathbf{x}_j, \mathbf{x}_k) = \begin{cases} 1 & \text{if} \quad c_i(\mathbf{x}_j) = c_i(\mathbf{x}_k), \\ 0 & \text{if} \quad c_i(\mathbf{x}_j) \neq c_i(\mathbf{x}_k). \end{cases}$$

Where $S(\tilde{\mathbf{x}}_j, \tilde{\mathbf{x}}_k)$ denotes the similarity between two samples based on the behaviour knowledge space method (BKS). Instances with similarity lower than a predefined threshold are removed from the region of competence. The competence level of the base classifiers are estimated as their classification accuracy in the final region of competence.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select (*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

Modified Local Accuracy (MLA)

class deslib.dcs.mla.**MLA**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *selection_method='best'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Modified Local Accuracy (MLA).

Similar to the LCA technique. The only difference is that the output of each base classifier is weighted by the distance between the test sample and each pattern in the region of competence for the estimation of the classifiers competences. Only the classifier that achieved the highest competence level is select to predict the label of the test sample x.

The MLA method selects the base classifier presenting the highest competence level. In a case where more than one base classifier achieves the same competence level, the one that was evaluated first is selected. The selection methodology can be modified by changing the hyper-parameter selection_method.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Woods, Kevin, W. Philip Kegelmeyer, and Kevin Bowyer. "Combination of multiple classifiers using local accuracy estimates." IEEE transactions on pattern analysis and machine intelligence 19.4 (1997): 405-410.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i for the classification of the query sample using the Modified Local Accuracy (MLA) method.

The competence level of the base classifiers is estimated by its classification accuracy taking into account only the samples belonging to a given class w_l.In this case, w_l is the class predicted by the base classifier c_i, for the query sample. This method also weights the influence of each training sample according to its Euclidean distance to the query instance. The closest samples have a higher influence in the computation

of the competence level. The competence level estimate is represented by the following equation:

$$\delta_{i,j} = \sum_{k=1}^{K} P(\omega_l \mid \mathbf{x}_k \in \omega_l, c_i) W_k$$

where $\delta_{i,j}$ represents the competence level of c_i for the classification of query.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

Overall Local Accuracy (OLA)

class deslib.dcs.ola.**OLA**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *selection_method='best'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Overall Classifier Accuracy (OLA).

The OLA method evaluates the competence level of each individual classifiers and select the most competent one to predict the label of each test sample x. The competence of each base classifier is calculated as its classification accuracy in the neighborhood of x (region of competence).

The OLA method selects the base classifier presenting the highest competence level. In a case where more than one base classifier achieves the same competence level, the one that was evaluated first is selected. The selection methodology can be modified by changing the hyper-parameter selection_method.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Woods, Kevin, W. Philip Kegelmeyer, and Kevin Bowyer. "Combination of multiple classifiers using local accuracy estimates." IEEE transactions on pattern analysis and machine intelligence 19.4 (1997): 405-410.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence level of each base classifier c_i for the classification of the query sample.

The competences for each base classifier c_i is estimated by its classification accuracy considering the k-Nearest Neighbors (region of competence). The competence level estimate is represented by the following equation:

$$\delta_{i,j} = \frac{1}{K} \sum_{k=1}^{K} P(\omega_l \mid \mathbf{x}_k \in \omega_l, c_i)$$

where $\delta_{i,j}$ represents the competence level of c_i for the classification of query.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

Modified Rank

class deslib.dcs.rank.**Rank**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *selection_method='best'*, *diff_thresh=0.1*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Modified Classifier Rank.

The modified classifier rank method evaluates the competence level of each individual classifiers and select the most competent one to predict the label of each test sample x. The competence of each base classifier is

calculated as the number of correctly classified samples, starting from the closest neighbor of x. The classifier with the highest number of correctly classified samples is considered the most competent.

The Rank method selects the base classifier presenting the highest competence level. In a case where more than one base classifier achieves the same competence level, the one that was evaluated first is selected. The selection methodology can be modified by changing the hyper-parameter selection_method.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

selection_method [String (Default = "best")] Determines which method is used to select the base classifier after the competences are estimated.

diff_thresh [float (Default = 0.1)] Threshold to measure the difference between the competence level of the base classifiers for the random and diff selection schemes. If the difference is lower than the threshold, their performance are considered equivalent.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Woods, Kevin, W. Philip Kegelmeyer, and Kevin Bowyer. "Combination of multiple classifiers using local accuracy estimates." IEEE transactions on pattern analysis and machine intelligence 19.4 (1997): 405-410.

M. Sabourin, A. Mitiche, D. Thomas, G. Nagy, Classifier combination for handprinted digit recognition, International Conference on Document Analysis and Recognition (1993) 163–166.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence level of each base classifier c_i for the classification of the query sample using the modified ranking scheme. The rank of the base classifier is estimated by the number of consecutive correctly classified samples in the defined region of competence.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for the test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select (*competences*)

Select the most competent classifier for the classification of the query sample given the competence level estimates. Four selection schemes are available.

Best : The base classifier with the highest competence level is selected. In cases where more than one base classifier achieves the same competence level, the one with the lowest index is selected. This method is the standard for the LCA, OLA, MLA techniques.

Diff : Select the base classifier that is significantly better than the others in the pool (when the difference between its competence level and the competence level of the other base classifiers is higher than a predefined threshold). If no base classifier is significantly better, the base classifier is selected randomly among the member with equivalent competence level.

Random : Selects a random base classifier among all base classifiers that achieved the same competence level.

ALL : all base classifiers with the max competence level estimates are selected (note that in this case the DCS technique becomes a DES technique).

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape [n_samples]] Indices of the selected base classifier for each sample. If the selection_method is set to 'all', a boolean matrix is returned, containing True for the selected base classifiers, otherwise false.

3.2.2 Dynamic Ensemble Selection (DES)

Dynamic ensemble selection strategies refer to techniques that select an ensemble of classifier rather than a single one. All base classifiers that attain a minimum competence level are selected to compose the ensemble of classifiers.

The *deslib.des* provides a set of key dynamic ensemble selection algorithms (DES).

META-DES

class deslib.des.meta_des.**METADES**(*pool_classifiers=None*, *meta_classifier=None*, *k=7*, *Kp=5*, *Hc=1.0*, *selection_threshold=0.5*, *mode='selection'*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Meta learning for dynamic ensemble selection (META-DES).

The META-DES framework is based on the assumption that the dynamic ensemble selection problem can be considered as a meta-problem. This meta-problem uses different criteria regarding the behavior of a base classifier c_i, in order to decide whether it is competent enough to classify a given test sample.

The framework performs a meta-training stage, in which, the meta-features are extracted from each instance belonging to the training and the dynamic selection dataset (DSEL). Then, the extracted meta-features are used to train the meta-classifier λ. The meta-classifier is trained to predict whether or not a base classifier c_i is competent enough to classify a given input sample.

When an unknown sample is presented to the system, the meta-features for each base classifier c_i in relation to the input sample are calculated and presented to the meta-classifier. The meta-classifier estimates the competence level of the base classifier c_i for the classification of the query sample. Base classifiers with competence level higher than a pre-defined threshold are selected. If no base classifier is selected, the whole pool is used for classification.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

meta_classifier [sklearn.estimator (Default = None)] Classifier model used for the meta-classifier. If None, a Multinomial naive Bayes classifier is used.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

Kp [int (Default = 5)] Number of output profiles used to estimate the competence of the base classifiers.

Hc [float (Default = 1.0)] Sample selection threshold.

selection_threshold [float(Default = 0.5)] Threshold used to select the base classifier. Only the base classifiers with competence level higher than the selection_threshold are selected to compose the ensemble.

mode [String (Default = "selection")] Determines the mode of META-des that is used (selection, weighting or hybrid).

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance,

random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Cruz, R.M., Sabourin, R., Cavalcanti, G.D. and Ren, T.I., 2015. META-DES: A dynamic ensemble selection framework using meta-learning. Pattern Recognition, 48(5), pp.1925-1935.

Cruz, R.M., Sabourin, R. and Cavalcanti, G.D., 2015, July. META-des. H: a dynamic ensemble selection technique using meta-learning and a dynamic weighting approach. In Neural Networks (IJCNN), 2015 International Joint Conference on (pp. 1-8).

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence_from_proba(*query*, *neighbors*, *probabilities*, *distances=None*)

Estimate the competence of each base classifier c_i the classification of the query sample. This method received an array with the pre-calculated probability estimates for each query.

First, the meta-features of each base classifier c_i for the classification of the query sample are estimated. These meta-features are passed down to the meta-classifier λ for the competence level estimation.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

probabilities [array of shape (n_samples, n_classifiers, n_classes)] Probabilities estimates obtained by each each base classifier for each query sample.

Returns

competences [array of shape (n_samples, n_classifiers)] The competence level estimated for each base classifier and test example.

fit (*X*, *y*)
Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS method.

This method also extracts the meta-features and trains the meta-classifier λ if the meta-classifier was not yet trained.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)
Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)
Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

score (*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select (*competences*)
Selects the base classifiers that obtained a competence level higher than the predefined threshold defined in self.selection_threshold.

Parameters

competences [array of shape (n_samples, n_classifiers)] The competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

DES Clustering

class deslib.des.des_clustering.**DESClustering**(*pool_classifiers=None*, *clustering=None*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *pct_accuracy=0.5*, *pct_diversity=0.33*, *more_diverse=True*, *metric_diversity='DF'*, *metric_performance='accuracy_score'*, *n_clusters=5*, *random_state=None*, *DSEL_perc=0.5*, *n_jobs=-1*)

Dynamic ensemble selection-Clustering (DES-Clustering).

This method selects an ensemble of classifiers taking into account the accuracy and diversity of the base classifiers. The K-means algorithm is used to define the region of competence. For each cluster, the N most accurate classifiers are first selected. Then, the J more diverse classifiers from the N most accurate classifiers are selected to compose the ensemble.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

clustering [sklearn.cluster (Default = None)] The clustering model used to estimate the region of competence. If None, a KMeans with K = 5 is used.

pct_accuracy [float (Default = 0.5)] Percentage of base classifiers selected based on accuracy

pct_diversity [float (Default = 0.33)] Percentage of base classifiers selected based on diversity

more_diverse [Boolean (Default = True)] Whether we select the most or the least diverse classifiers to add to the pre-selected ensemble

metric_diversity [String (Default = 'df')] Metric used to estimate the diversity of the base classifiers. Can be either the double fault (df), Q-statistics (Q), or error correlation.

metric_performance [String (Default = 'accuracy_score')] Metric used to estimate the performance of a base classifier on a cluster. Can be either any metric from sklearn.metrics.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Soares, R. G., Santana, A., Canuto, A. M., & de Souto, M. C. P. "Using accuracy and more_diverse to select classifiers to build ensembles." International Joint Conference on Neural Networks (IJCNN)., 2006.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *predictions=None*)

Get the competence estimates of each base classifier c_i for the classification of the query sample.

In this case, the competences were already pre-calculated for each cluster. So this method computes the nearest cluster and get the pre-calculated competences of the base classifiers for the corresponding cluster.

Parameters

query [array of shape (n_samples, n_features)] The query sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array = [n_samples, n_classifiers]] The competence level estimated for each base classifier.

fit(*X*, *y*)

Train the DS model by setting the Clustering algorithm and pre-processing the information required to apply the DS methods.

First the data is divided into K clusters. Then, for each cluster, the N most accurate classifiers are first selected. Then, the J more diverse classifiers from the N most accurate classifiers are selected to compose the ensemble of the corresponding cluster. An ensemble of classifiers is assigned to each of the K clusters.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*query*)

Select an ensemble with the most accurate and most diverse classifier for the classification of the query.

The ensemble for each cluster was already pre-calculated in the fit method. So, this method calculates the closest cluster, and returns the ensemble associated to this cluster.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

Returns

selected_classifiers [array of shape = [n_samples, self.k]] Indices of the selected base classifier for each test example.

Dynamic Ensemble Selection performance (DES-P)

class deslib.des.des_p.**DESP**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *mode='selection'*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Dynamic ensemble selection-Performance(DES-P).

This method selects all base classifiers that achieve a classification performance, in the region of competence, that is higher than the random classifier (RC). The performance of the random classifier is defined by RC = 1/L, where L is the number of classes in the problem. If no base classifier is selected, the whole pool is used for classification.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

mode [String (Default = "selection")] Whether the technique will perform dynamic selection, dynamic weighting or an hybrid approach for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method..

References

Woloszynski, Tomasz, et al. "A measure of competence based on random classification for dynamic ensemble selection." Information Fusion 13.3 (2012): 207-213.

Woloszynski, Tomasz, and Marek Kurzynski. "A probabilistic model of classifier competence for dynamic ensemble selection." Pattern Recognition 44.10 (2011): 2656-2668.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence of each base classifier c_i for the classification of the query sample base on its local performance.

$$\delta_{i,j} = \hat{P}(c_i \mid \theta_j) - \frac{1}{L}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select (*competences*)

Selects all base classifiers that obtained a local classification accuracy higher than the Random Classifier. The performance of the random classifier is denoted 1/L, where L is the number of classes in the problem.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

DES-KNN

class deslib.des.des_knn.**DESKNN**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *pct_accuracy=0.5*, *pct_diversity=0.3*, *more_diverse=True*, *metric='DF'*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Dynamic ensemble Selection KNN (DES-KNN).

This method selects an ensemble of classifiers taking into account the accuracy and diversity of the base classifiers. The k-NN algorithm is used to define the region of competence. The N most accurate classifiers in the region of competence are first selected. Then, the J more diverse classifiers from the N most accurate classifiers are selected to compose the ensemble.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

pct_accuracy [float (Default = 0.5)] Percentage of base classifiers selected based on accuracy

pct_diversity [float (Default = 0.3)] Percentage of base classifiers selected based n diversity

more_diverse [Boolean (Default = True)] Whether we select the most or the least diverse classifiers to add to the pre-selected ensemble

metric [String (Default = 'df')] Metric used to estimate the diversity of the base classifiers. Can be either the double fault (df), Q-statistics (Q), or error correlation.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

KNNE available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Soares, R. G., Santana, A., Canuto, A. M., & de Souto, M. C. P. "Using accuracy and more_diverse to select classifiers to build ensembles." International Joint Conference on Neural Networks (IJCNN)., 2006.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence level of each base classifier c_i for the classification of the query sample.

The competence is estimated using the accuracy and diversity criteria. First the classification accuracy of the base classifiers in the region of competence is estimated. Then the diversity of the base classifiers is estimated.

The method returns two arrays: One containing the accuracy and the other the diversity of each base classifier.

Parameters

query [array of shape (n_samples, n_features)] The query sample.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

accuracy [array of shape = [n_samples, n_classifiers}] Local Accuracy estimates (competences) of the base classifiers for all query samples.

diversity [array of shape = [n_samples, n_classifiers}] Average pairwise diversity of each base classifiers for all test examples.

Notes

This technique uses both the accuracy and diversity information to perform dynamic selection. For this reason the function returns a dictionary containing these two values instead of a single ndarray containing the competence level estimates for each base classifier.

fit (*X*, *y*)
Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS method.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)
Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)
Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

score (*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select (*accuracy*, *diversity*)
Select an ensemble containing the N most accurate ant the J most diverse classifiers for the classification of the query sample.

Parameters

accuracy [array of shape (n_samples, n_classifiers)] Local Accuracy estimates (competence) of each base classifiers.

diversity [array of shape (n_samples, n_classifiers)] Average pairwise diversity of each base classifiers.

Returns

selected_classifiers [array of shape = [n_samples, self.J]] Array containing the indices of the J selected base classifier for each test example.

k-Nearest Output Profiles (KNOP)

class deslib.des.knop.**KNOP**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

k-Nearest Output Profiles (KNOP).

This method selects all classifiers that correctly classified at least one sample belonging to the region of competence of the query sample. In this case, the region of competence is estimated using the decisions of the base classifier (output profiles). Thus, the similarity between the query and the validation samples are measured in the decision space rather than the feature space. Each selected classifier has a number of votes equals to the number of samples in the region of competence that it predicts the correct label. The votes obtained by all base classifiers are aggregated to obtain the final ensemble decision.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`

- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Cavalin, Paulo R., Robert Sabourin, and Ching Y. Suen. "LoGID: An adaptive framework combining local and global incremental learning for dynamic selection of ensembles of HMMs." Pattern Recognition 45.9 (2012): 3544-3556.

Cavalin, Paulo R., Robert Sabourin, and Ching Y. Suen. "Dynamic selection approaches for multiple classifier systems." Neural Computing and Applications 22.3-4 (2013): 673-688.

Ko, Albert HR, Robert Sabourin, and Alceu Souza Britto Jr. "From dynamic classifier selection to dynamic ensemble selection." Pattern Recognition 41.5 (2008): 1718-1731.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence_from_proba(*query*, *probabilities*, *neighbors=None*, *distances=None*)

The competence of the base classifiers is simply estimated as the number of samples in the region of competence that it correctly classified. This method received an array with the pre-calculated probability estimates for each query.

This information is later used to determine the number of votes obtained for each base classifier.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

probabilities [array of shape (n_samples, n_classifiers, n_classes)] Probabilities estimates obtained by each each base classifier for each query sample.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-process the information required to apply the DS methods. In this case, the scores of the base classifiers for the dynamic selection dataset (DSEL) are pre-calculated to transform each sample in DSEL into an output profile.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the base classifiers for the classification of the query sample.

Each base classifier can be selected more than once. The number of times a base classifier is selected (votes) is equals to the number of samples it correctly classified in the region of competence.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

k-Nearest Oracle-Eliminate (KNORA-E)

class deslib.des.knora_e.**KNORAE**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

k-Nearest Oracles Eliminate (KNORA-E).

This method searches for a local Oracle, which is a base classifier that correctly classify all samples belonging to the region of competence of the test sample. All classifiers with a perfect performance in the region of competence are selected (local Oracles). In the case that no classifier achieves a perfect accuracy, the size of the competence region is reduced (by removing the farthest neighbor) and the performance of the classifiers are re-evaluated. The outputs of the selected ensemble of classifiers is combined using the majority voting scheme. If no base classifier is selected, the whole pool is used for classification.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Ko, Albert HR, Robert Sabourin, and Alceu Souza Britto Jr. "From dynamic classifier selection to dynamic ensemble selection." Pattern Recognition 41.5 (2008): 1718-1731.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

Estimate the competence of the base classifiers. In the case of the KNORA-E technique, the classifiers are only considered competent when they achieve a 100% accuracy in the region of competence. For each base, we estimate the maximum size of the region of competence that it is a local oracle. The competence level estimate is then the maximum size of the region of competence that the corresponding base classifier is considered a local Oracle.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Selects all base classifiers that obtained a local accuracy of 100% in the region of competence (i.e., local oracle). In the case that no base classifiers obtain 100% accuracy, the size of the region of competence is reduced and the search for the local oracle is restarted.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

Notes

Instead of re-applying the method several times (reducing the size of the region of competence), we compute the number of consecutive correct classification of each base classifier starting from the closest neighbor to the more distant in the estimate_competence function. The number of consecutive correct classification represents the size of the region of competence in which the corresponding base classifier is an Local Oracle. Then, we select all base classifiers with the maximum value for the number of consecutive correct classification. This speed up the selection process.

k-Nearest Oracle Union (KNORA-U)

class deslib.des.knora_u.**KNORAU**(*pool_classifiers=None*, *k=7*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

k-Nearest Oracles Union (KNORA-U).

This method selects all classifiers that correctly classified at least one sample belonging to the region of competence of the query sample. Each selected classifier has a number of votes equals to the number of samples in the region of competence that it predicts the correct label. The votes obtained by all base classifiers are aggregated to obtain the final ensemble decision.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

  `KNNE` available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Ko, Albert HR, Robert Sabourin, and Alceu Souza Britto Jr. "From dynamic classifier selection to dynamic ensemble selection." Pattern Recognition 41.5 (2008): 1718-1731.

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

The competence of the base classifiers is simply estimated as the number of samples in the region of competence that it correctly classified.

This information is later used to determine the number of votes obtained for each base classifier.

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select the base classifiers for the classification of the query sample.

Each base classifier can be selected more than once. The number of times a base classifier is selected (votes) is equals to the number of samples it correctly classified in the region of competence.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

DES Multiclass Imbalance (DES-MI)

class deslib.des.des_mi.**DESMI**(*pool_classifiers=None*, *k=7*, *pct_accuracy=0.4*, *alpha=0.9*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *random_state=None*, *knn_classifier='knn'*, *knne=False*, *DSEL_perc=0.5*, *n_jobs=-1*)

Dynamic ensemble Selection for multi-class imbalanced datasets (DES-MI).

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

alpha [float (Default = 0.9)] Scaling coefficient to regulate the weight value

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn

KNNE available on *deslib.utils.knne*

- 'faiss' will use Facebook's Faiss similarity search through the class FaissKNNClassifier
- None, will use sklearn KNeighborsClassifier.

knne [bool (Default=False)] Whether to use K-Nearest Neighbor Equality (KNNE) for the region of competence estimation.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

García, S.; Zhang, Z.-L.; Altalhi, A.; Alshomrani, S. & Herrera, F. "Dynamic ensemble selection for multi-class imbalanced datasets." Information Sciences, 2018, 445-446, 22 - 37

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances=None*, *predictions=None*)

estimate the competence level of each base classifier c_i for the classification of the query sample. Returns a ndarray containing the competence level of each base classifier.

The competence is estimated using the accuracy criteria. The accuracy is estimated by the weighted results of classifiers who correctly classify the members in the competence region. The weight of member x_i is related to the number of samples of the same class of x_i in the training dataset. For detail, please see the first reference, Algorithm 2.

Parameters

query [array of shape (n_samples, n_features)] The query sample.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

accuracy [array of shape = [n_samples, n_classifiers}] Local Accuracy estimates (competences) of the base classifiers for all query samples.

fit(*X*, *y*)

Prepare the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Select an ensemble containing the N most accurate classifiers for the classification of the query sample.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence estimates of each base classifiers for all query samples.

Returns

selected_classifiers [array of shape = [n_samples, self.N]] Matrix containing the indices of the N selected base classifier for each test example.

Probabilistic

class deslib.des.probabilistic.**BaseProbabilistic**(*pool_classifiers=None*, *k=None*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *mode='selection'*, *selection_threshold=None*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *n_jobs=-1*)

Base class for a DS method based on the potential function model. All DS methods based on the Potential function should inherit from this class.

Warning: This class should not be used directly. Use derived classes instead.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i using the source of competence C_{src} and the potential function model. The source of competence C_{src} for all data points in DSEL is already pre-computed in the fit() steps.

$$\delta_{i,j} = \frac{\sum_{k=1}^{N} C_{src}\, exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}{exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods. In the case of probabilistic techniques, the source of competence (C_src) is calculated for each data point in DSEL in order to speed up the process during the testing phases.

C_src is estimated with the source_competence() function that is overridden by each DS method based on this paradigm.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

static potential_func(*dist*)

Gaussian potential function to decrease the influence of the source of competence as the distance between

$\mathbf{x}_k$ and the query $\mathbf{x}_q$ increases. The function is computed using the following equation:

$$potential = exp(-dist(\mathbf{x}_k, \mathbf{x}_q)^2)$$

where dist represents the Euclidean distance between $\mathbf{x}_k$ and $\mathbf{x}_q$

Parameters

dist [array of shape = [self.n_samples]] distance between the corresponding sample to the query

Returns

The result of the potential function for each value in (dist)

select(*competences*)

Selects the base classifiers that obtained a competence level higher than the predefined threshold. In this case, the threshold indicates the competence of the random classifier.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

source_competence()

Method used to estimate the source of competence at each data point.

Each DS technique based on this paradigm should define its computation of C_src

Returns

C_src [array of shape (n_samples, n_classifiers)] The competence source for each base classifier at each data point.

Randomized Reference Classifier (RRC)

class deslib.des.probabilistic.**RRC**(*pool_classifiers=None*, *k=None*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *mode='selection'*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *n_jobs=-1*)

DES technique based on the Randomized Reference Classifier method (DES-RRC).

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

mode [String (Default = "selection")] Whether the technique will perform dynamic selection, dynamic weighting or an hybrid approach for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn
- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Woloszynski, Tomasz, and Marek Kurzynski. "A probabilistic model of classifier competence for dynamic ensemble selection." Pattern Recognition 44.10 (2011): 2656-2668.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i using the source of competence C_{src} and the potential function model. The source of competence C_{src} for all data points in DSEL is already pre-computed in the fit() steps.

$$\delta_{i,j} = \frac{\sum_{k=1}^{N} C_{src}\, exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}{exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods. In the case of probabilistic techniques, the source of competence (C_src) is calculated for each data point in DSEL in order to speed up the process during the testing phases.

C_src is estimated with the source_competence() function that is overridden by each DS method based on this paradigm.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select (*competences*)

Selects the base classifiers that obtained a competence level higher than the predefined threshold. In this case, the threshold indicates the competence of the random classifier.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

source_competence()

Calculates the source of competence using the randomized reference classifier (RRC) method.

The source of competence C_src at the validation point $\mathbf{x}_k$ calculated using the probabilistic model based on the supports obtained by the base classifier and randomized reference classifier (RRC) model. The probabilistic modeling of the classifier competence is calculated using the ccprmod function.

Returns

C_src [array of shape (n_samples, n_classifiers)] The competence source for each base classifier at each data point.

DES-Kullback Leibler

class deslib.des.probabilistic.**DESKL**(*pool_classifiers=None*, *k=None*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *mode='selection'*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *n_jobs=-1*)

Dynamic Ensemble Selection-Kullback-Leibler divergence (DES-KL).

This method estimates the competence of the classifier from the information theory perspective. The competence of the base classifiers is calculated as the KL divergence between the vector of class supports produced by the base classifier and the outputs of a random classifier (RC) RC = 1/L, L being the number of classes in the problem. Classifiers with a competence higher than the competence of the random classifier is selected.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

mode [String (Default = "selection")] Whether the technique will perform dynamic selection, dynamic weighting or an hybrid approach for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn
- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Woloszynski, Tomasz, et al. "A measure of competence based on random classification for dynamic ensemble selection." Information Fusion 13.3 (2012): 207-213.

Woloszynski, Tomasz, and Marek Kurzynski. "A probabilistic model of classifier competence for dynamic ensemble selection." Pattern Recognition 44.10 (2011): 2656-2668.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i using the source of competence C_{src} and the potential function model. The source of competence C_{src} for all data points in DSEL is already pre-computed in the fit() steps.

$$\delta_{i,j} = \frac{\sum_{k=1}^{N} C_{src}\, exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}{exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods. In the case of probabilistic techniques, the source of competence (C_src) is calculated for each data point in DSEL in order to speed up the process during the testing phases.

C_src is estimated with the source_competence() function that is overridden by each DS method based on this paradigm.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Selects the base classifiers that obtained a competence level higher than the predefined threshold. In this case, the threshold indicates the competence of the random classifier.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

source_competence()

Calculates the source of competence using the KL divergence method.

The source of competence C_src at the validation point $\mathbf{x}_k$ is calculated by the KL divergence between the vector of class supports produced by the base classifier and the outputs of a random classifier (RC) RC = 1/L, L being the number of classes in the problem. The value of C_src is negative if the base classifier misclassified the instance $\mathbf{x}_k$.

Returns

C_src [array of shape (n_samples, n_classifiers)] The competence source for each base classifier at each data point.

DES-Minimum Difference

class deslib.des.probabilistic.**MinimumDifference**(*pool_classifiers=None*, *k=None*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *mode='selection'*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *n_jobs=-1*)

Computes the competence level of the classifiers based on the difference between the support obtained by each class. The competence level at a data point $\mathbf{x}_k$ is equal to the minimum difference between the support obtained to the correct class and the support obtained for different classes.

The influence of each sample xk is defined according to a Gaussian function model[2]. Samples that are closer to the query have a higher influence in the competence estimation.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

mode [String (Default = "selection")] Whether the technique will perform dynamic selection, dynamic weighting or an hybrid approach for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn
- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

[1] B. Antosik, M. Kurzynski, New measures of classifier competence – heuristics and application to the design of multiple classifier systems., in: Computer recognition systems 4., 2011, pp. 197–206.

[2] Woloszynski, Tomasz, and Marek Kurzynski. "A probabilistic model of classifier competence for dynamic ensemble selection." Pattern Recognition 44.10 (2011): 2656-2668.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i using the source of competence C_{src} and the potential function model. The source of competence C_{src} for all data points in DSEL is already pre-computed in the fit() steps.

$$\delta_{i,j} = \frac{\sum_{k=1}^{N} C_{src}\, exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}{exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods. In the case of probabilistic techniques, the source of competence (C_src) is calculated for each data point in DSEL in order to speed up the process during the testing phases.

C_src is estimated with the source_competence() function that is overridden by each DS method based on this paradigm.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Selects the base classifiers that obtained a competence level higher than the predefined threshold. In this case, the threshold indicates the competence of the random classifier.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

source_competence()

Calculates the source of competence using the Minimum Difference method.

The source of competence C_src at the validation point $\mathbf{x}_k$ calculated by the Minimum Difference between the supports obtained to the correct class and the support obtained by the other classes

Returns

C_src [array of shape (n_samples, n_classifiers)] The competence source for each base classifier at each data point.

DES-Exponential

class deslib.des.probabilistic.**Exponential**(*pool_classifiers=None*, *k=None*, *DFP=False*, *safe_k=None*, *with_IH=False*, *IH_rate=0.3*, *mode='selection'*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *n_jobs=-1*)

The source of competence C_src at the validation point $\mathbf{x}_k$ is a product of two factors: The absolute value of the competence and the sign. The value of the source competence is inverse proportional to the normalized entropy of its supports vector. The sign of competence is simply determined by correct/incorrect classification of $\mathbf{x}_k$ [1].

The influence of each sample $\mathbf{x}_k$ is defined according to a Gaussian function model[2]. Samples that are closer to the query have a higher influence in the competence estimation.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

mode [String (Default = "selection")] Whether the technique will perform dynamic selection, dynamic weighting or an hybrid approach for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn
- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

[1] B. Antosik, M. Kurzynski, New measures of classifier competence – heuristics and application to the design of multiple classifier systems., in: Computer recognition systems 4., 2011, pp. 197–206.

[2] Woloszynski, Tomasz, and Marek Kurzynski. "A probabilistic model of classifier competence for dynamic ensemble selection." Pattern Recognition 44.10 (2011): 2656-2668.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i using the source of competence C_{src} and the potential function model. The source of competence C_{src} for all data points in DSEL is already pre-computed in the fit() steps.

$$\delta_{i,j} = \frac{\sum_{k=1}^{N} C_{src}\, exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}{exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit(*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods. In the case of probabilistic techniques, the source of competence (C_src) is calculated for each data point in DSEL in order to speed up the process during the testing phases.

C_src is estimated with the source_competence() function that is overridden by each DS method based on this paradigm.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Selects the base classifiers that obtained a competence level higher than the predefined threshold. In this case, the threshold indicates the competence of the random classifier.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

source_competence()

The source of competence C_src at the validation point $\mathbf{x}_k$ is a product of two factors: The absolute value of the competence and the sign. The value of the source competence is inverse proportional to the normalized entropy of its supports vector.The sign of competence is simply determined by correct/incorrect classification of the instance $\mathbf{x}_k$.

Returns

C_src [array of shape (n_samples, n_classifiers)] The competence source for each base classifier at each data point.

DES-Logarithmic

class deslib.des.probabilistic.**Logarithmic**(*pool_classifiers=None*, *k=None*, *DFP=False*, *with_IH=False*, *safe_k=None*, *IH_rate=0.3*, *mode='selection'*, *random_state=None*, *knn_classifier='knn'*, *DSEL_perc=0.5*, *n_jobs=-1*)

This method estimates the competence of the classifier based on the logarithmic difference between the supports obtained by the base classifier.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

k [int (Default = 7)] Number of neighbors used to estimate the competence of the base classifiers.

DFP [Boolean (Default = False)] Determines if the dynamic frienemy pruning is applied.

with_IH [Boolean (Default = False)] Whether the hardness level of the region of competence is used to decide between using the DS algorithm or the KNN for classification of a given query sample.

safe_k [int (default = None)] The size of the indecision region.

IH_rate [float (default = 0.3)] Hardness threshold. If the hardness level of the competence region is lower than the IH_rate the KNN classifier is used. Otherwise, the DS algorithm is used for classification.

mode [String (Default = "selection")] Whether the technique will perform dynamic selection, dynamic weighting or an hybrid approach for classification.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

knn_classifier [{'knn', 'faiss', None} (Default = 'knn')] The algorithm used to estimate the region of competence:

- 'knn' will use `KNeighborsClassifier` from sklearn
- 'faiss' will use Facebook's Faiss similarity search through the class `FaissKNNClassifier`
- None, will use sklearn `KNeighborsClassifier`.

DSEL_perc [float (Default = 0.5)] Percentage of the input data used to fit DSEL. Note: This parameter is only used if the pool of classifier is None or unfitted.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

B. Antosik, M. Kurzynski, New measures of classifier competence – heuristics and application to the design of multiple classifier systems., in: Computer recognition systems 4., 2011, pp. 197–206.

T.Woloszynski, M. Kurzynski, A measure of competence based on randomized reference classifier for dynamic ensemble selection, in: International Conference on Pattern Recognition (ICPR), 2010, pp. 4194–4197.

estimate_competence(*query*, *neighbors*, *distances*, *predictions=None*)

estimate the competence of each base classifier c_i using the source of competence C_{src} and the potential function model. The source of competence C_{src} for all data points in DSEL is already pre-computed in the fit() steps.

$$\delta_{i,j} = \frac{\sum_{k=1}^{N} C_{src}\, exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}{exp(-d(\mathbf{x}_k, \mathbf{x}_q)^2)}$$

Parameters

query [array of shape (n_samples, n_features)] The test examples.

neighbors [array of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors according for each test sample.

distances [array of shape (n_samples, n_neighbors)] Distances of the k nearest neighbors according for each test sample.

predictions [array of shape (n_samples, n_classifiers)] Predictions of the base classifiers for all test examples.

Returns

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

fit (*X*, *y*)

Train the DS model by setting the KNN algorithm and pre-processing the information required to apply the DS methods. In the case of probabilistic techniques, the source of competence (C_src) is calculated for each data point in DSEL in order to speed up the process during the testing phases.

C_src is estimated with the source_competence() function that is overridden by each DS method based on this paradigm.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict (*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_labels [array of shape (n_samples)] Predicted class label for each sample in X.

predict_proba (*X*)

Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

select(*competences*)

Selects the base classifiers that obtained a competence level higher than the predefined threshold. In this case, the threshold indicates the competence of the random classifier.

Parameters

competences [array of shape (n_samples, n_classifiers)] Competence level estimated for each base classifier and test example.

Returns

selected_classifiers [array of shape (n_samples, n_classifiers)] Boolean matrix containing True if the base classifier is selected, False otherwise.

source_competence()

The source of competence C_src at the validation point $\mathbf{x}_k$ is calculated by logarithm function in the support obtained by the base classifier.

Returns

C_src [array of shape (n_samples, n_classifiers)] The competence source for each base classifier at each data point.

3.2.3 Static ensembles

This module provides the implementation of static ensemble techniques that are usually used as a baseline for the comparison of DS methods: Single Best (SB), Static Selection (SS), Stacked classifier and Oracle.

The *deslib.static* provides a set of static ensemble methods which are often used as a baseline to compare the performance of dynamic selection algorithms.

Oracle

class deslib.static.oracle.**Oracle**(*pool_classifiers=None*, *random_state=None*, *n_jobs=-1*)

Abstract method that always selects the base classifier that predicts the correct label if such classifier exists. This method is often used to measure the upper-limit performance that can be achieved by a dynamic classifier selection technique. It is used as a benchmark by several dynamic selection algorithms

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Kuncheva, Ludmila I. "A theoretical study on six classifier fusion strategies." IEEE Transactions on Pattern Analysis & Machine Intelligence, (2002): 281-286.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

fit (*X*, *y*)
Fit the model according to the given training data.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict (*X*, *y*)
Prepare the labels using the Oracle model.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified

y [array of shape (n_samples)] Class labels of each sample in X.

Returns

predicted_labels [array of shape (n_samples)] Predicted class for each sample in X.

predict_proba (*X*, *y*)
Estimates the posterior probabilities for each class for each sample in X.

Note that as the Oracle is the ideal classifier selection, the classifier that estimate the highest probability for the correct class is the selected one.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified.

y [array of shape (n_samples)] Class labels of each sample in X.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Posterior probabilities estimates for each class.

score (*X*, *y*, *sample_weights=None*)
Prepare the labels using the Oracle model.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified.

y [array of shape (n_samples)] Class labels of each sample in X.

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

Returns

accuracy [float] Classification accuracy of the Oracle model.

Single Best

class deslib.static.single_best.**SingleBest**(*pool_classifiers=None*, *scoring=None*, *random_state=None*, *n_jobs=-1*)

Classification method that selects the classifier in the pool with highest score to be used for classification. Usually, the performance of the single best classifier is estimated based on the validation data.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

scoring [string, callable (default = None)] A single string or a callable to evaluate the predictions on the validation set.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

Kuncheva, Ludmila I. Combining pattern classifiers: methods and algorithms. John Wiley & Sons, 2004.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

fit(*X*, *y*)

Fit the model by selecting the base classifier with the highest accuracy in the dataset. The single best classifier is kept in self.best_clf and its index is kept in self.best_clf_index.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

predict(*X*)

Predict the label of each sample in X and returns the predicted label.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified

Returns

predicted_labels [array of shape (n_samples)] Predicted class for each sample in X.

predict_proba(*X*)

Estimates the posterior probabilities for each class for each sample in X. The returned probability estimates for all classes are ordered by the label of classes.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified

Returns

predicted_proba [array of shape (n_samples, n_classes)] Posterior probabilities estimates for each class.

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

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

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

Static Selection

class deslib.static.static_selection.**StaticSelection**(*pool_classifiers=None*, *pct_classifiers=0.5*, *scoring=None*, *random_state=None*, *n_jobs=-1*)

Ensemble model that selects N classifiers with the best performance in a dataset

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict". If None, then the pool of classifiers is a bagging classifier.

scoring [string, callable (default = None)] A single string or a callable to evaluate the predictions on the validation set.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

pct_classifiers [float (Default = 0.5)] Percentage of base classifier that should be selected by the selection scheme.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

Kuncheva, Ludmila I. Combining pattern classifiers: methods and algorithms. John Wiley & Sons, 2004.

R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

fit (*X*, *y*)
Fit the static selection model by select an ensemble of classifier containing the base classifiers with highest accuracy in the given dataset.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

Returns

self [object] Returns self.

predict (*X*)
Predict the label of each sample in X and returns the predicted label.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified

Returns

predicted_labels [array of shape (n_samples)] Predicted class for each sample in X.

predict_proba (*X*)
Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

score (*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

Stacked Classifier

class deslib.static.stacked.**StackedClassifier**(*pool_classifiers=None*, *meta_classifier=None*, *passthrough=False*, *random_state=None*, *n_jobs=-1*)

A Stacking classifier.

Parameters

pool_classifiers [list of classifiers (Default = None)] The generated_pool of classifiers trained for the corresponding classification problem. Each base classifiers should support the method "predict" and "predict_proba". If None, then the pool of classifiers is a bagging classifier.

meta_classifier [object or None, optional (default=None)] Classifier model used to aggregate the output of the base classifiers. If None, a `LogisticRegression` classifier is used.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

passthrough [bool (default=False)] When False, only the predictions of estimators will be used as training data for the meta-classifier. When True, the meta-classifier is trained on the predictions as well as the original training data.

n_jobs [int, default=-1] The number of parallel jobs to run. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. Doesn't affect fit method.

References

Wolpert, David H. "Stacked generalization." Neural networks 5, no. 2 (1992): 241-259.

Kuncheva, Ludmila I. Combining pattern classifiers: methods and algorithms. John Wiley & Sons, 2004.

fit (*X*, *y*)
Fit the model by training a meta-classifier on the outputs of the base classifiers

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

predict (*X*)
Predict the label of each sample in X and returns the predicted label.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified

Returns

predicted_labels [array of shape (n_samples)] Predicted class for each sample in X.

predict_proba (*X*)
Predict the label of each sample in X and returns the predicted label.

Parameters

X [array of shape (n_samples, n_features)] The data to be classified

Returns

predicted_labels [array of shape (n_samples)] Predicted class for each sample in X.

score (*X*, *y*, *sample_weight=None*)
Return the mean accuracy on the given test data and labels.

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

Parameters

X [array-like of shape (n_samples, n_features)] Test samples.

y [array-like of shape (n_samples,) or (n_samples, n_outputs)] True labels for *X*.

sample_weight [array-like of shape (n_samples,), default=None] Sample weights.

Returns

score [float] Mean accuracy of `self.predict(X)` wrt. *y*.

3.2.4 Utils

Utility functions for ensemble methods such as diversity and aggregation methods.

The *deslib.util* This module includes various utilities. They are divided into four parts:

deslib.util.aggregation - Implementation of aggregation functions such as majority voting and averaging. Such functions can be applied to any list of classifiers.

deslib.util.diversity - Implementation of different measures of diversity between classifiers.

deslib.util.prob_functions - Functions to estimate the competence of a base classifier based on the probability estimates.

deslib.util.instance_hardness - Functions to measure the hardness level of a given instance

deslib.util.faiss_knn_wrapper - Wrapper for Facebook AI fast similarity search on GPU

deslib.util.datasets - Provides methods to generate synthetic data.

deslib.util.knne - Implementation of the K-Nearest Neighbors Equality technique

Diversity

This file contains the implementation of key diversity measures found in the ensemble literature:

- Double Fault
- Negative Double fault
- Q-statistics
- Ratio of errors

The implementation are made according to the specifications from the book "Combining Pattern Classifiers".

`deslib.util.diversity.`**`Q_statistic`**(*y*, *y_pred1*, *y_pred2*)

Calculates the Q-statistics diversity measure between a pair of classifiers. The Q value is in a range [-1, 1]. Classifiers that tend to classify the same object correctly will have positive values of Q, and Q = 0 for two independent classifiers.

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

Q [The q-statistic measure between two classifiers]

`deslib.util.diversity.`**`agreement_measure`**(*y*, *y_pred1*, *y_pred2*)

Calculates the agreement measure between a pair of classifiers. This measure is calculated by the frequency that both classifiers either obtained the correct or incorrect prediction for any given sample

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

agreement [The frequency at which both classifiers agrees]

deslib.util.diversity.**compute_pairwise_diversity**(*targets*, *prediction_matrix*, *diversity_func*)

Computes the pairwise diversity matrix.

Parameters

targets [array of shape (n_samples):] Class labels of each sample in X.

prediction_matrix [array of shape (n_samples, n_classifiers):] Predicted class labels for each classifier in the pool

diversity_func [Function] Function used to estimate the pairwise diversity

Returns

diversity [array of shape = [n_classifiers]] The average pairwise diversity matrix calculated for the pool of classifiers

deslib.util.diversity.**correlation_coefficient**(*y*, *y_pred1*, *y_pred2*)

Calculates the correlation between two classifiers using oracle outputs. Coefficient is a value in a range [-1, 1].

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

rho [The correlation coefficient measured between two classifiers]

deslib.util.diversity.**disagreement_measure**(*y*, *y_pred1*, *y_pred2*)

Calculates the disagreement measure between a pair of classifiers. This measure is calculated by the frequency that only one classifier makes the correct prediction.

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

disagreement [The frequency at which both classifiers disagrees]

deslib.util.diversity.**double_fault**(*y*, *y_pred1*, *y_pred2*)

Calculates the double fault (df) measure. This measure represents the probability that both classifiers makes the wrong prediction. A lower value of df means the base classifiers are less likely to make the same error. This measure must be minimized to increase diversity.

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

df [The double fault measure between two classifiers]

References

Giacinto, Giorgio, and Fabio Roli. "Design of effective neural network ensembles for image classification purposes." Image and Vision Computing 19.9 (2001): 699-707.

deslib.util.diversity.**negative_double_fault**(*y*, *y_pred1*, *y_pred2*)

The negative of the double fault measure. This measure should be maximized for a higher diversity.

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

df [The negative double fault measure between two classifiers]

References

Giacinto, Giorgio, and Fabio Roli. "Design of effective neural network ensembles for image classification purposes." Image and Vision Computing 19.9 (2001): 699-707.

deslib.util.diversity.**ratio_errors**(*y*, *y_pred1*, *y_pred2*)

Calculates Ratio of errors diversity measure between a pair of classifiers. A higher value means that the base classifiers are less likely to make the same errors. The ratio must be maximized for a higher diversity

Parameters

y [array of shape (n_samples):] class labels of each sample.

y_pred1 [array of shape (n_samples):] predicted class labels by the classifier 1 for each sample.

y_pred2 [array of shape (n_samples):] predicted class labels by the classifier 2 for each sample.

Returns

ratio [The q-statistic measure between two classifiers]

References

Aksela, Matti. "Comparison of classifier selection methods for improving committee performance." Multiple Classifier Systems (2003): 159-159.

Aggregation

This file contains the implementation of different aggregation functions to combine the outputs of the base classifiers to give the final decision.

deslib.util.aggregation.**average_combiner**(*classifier_ensemble*, *X*)
Ensemble combination using the Average rule.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**average_rule**(*predictions*)
Apply the average fusion rule to the predicted vector of class supports (predictions).

Parameters

predictions [np array of shape (n_samples, n_classifiers, n_classes)] Vector of class supports predicted by each base classifier for sample

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**majority_voting**(*classifier_ensemble*, *X*)
Apply the majority voting rule to predict the label of each sample in X.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**majority_voting_rule**(*votes*)
Applies the majority voting rule to the estimated votes.

Parameters

votes [array of shape (n_samples, n_classifiers),] The votes obtained by each classifier for each sample.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**maximum_combiner**(*classifier_ensemble*, *X*)
Ensemble combination using the Maximum rule.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**maximum_rule**(*predictions*)
Apply the product fusion rule to the predicted vector of class supports (predictions).

Parameters

predictions [np array of shape (n_samples, n_classifiers, n_classes)] Vector of class supports predicted by each base classifier for sample

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**median_combiner**(*classifier_ensemble*, *X*)
Ensemble combination using the Median rule.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**median_rule**(*predictions*)
Apply the product fusion rule to the predicted vector of class supports (predictions).

Parameters

predictions [np array of shape (n_samples, n_classifiers, n_classes)] Vector of class supports predicted by each base classifier for sample

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**minimum_combiner**(*classifier_ensemble*, *X*)
Ensemble combination using the Minimum rule.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**minimum_rule**(*predictions*)
Apply the product fusion rule to the predicted vector of class supports (predictions).

Parameters

predictions [np array of shape (n_samples, n_classifiers, n_classes)] Vector of class supports predicted by each base classifier for sample

Returns

list_proba [array of shape = [n_classifiers, n_samples, n_classes]] Probabilities predicted by each base classifier in the ensemble for all samples in X.

deslib.util.aggregation.**predict_proba_ensemble**(*classifier_ensemble*, *X*)
Estimates the posterior probabilities of the give ensemble for each sample in X.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_proba [array of shape (n_samples, n_classes)] Posterior probabilities estimates for each samples in X.

deslib.util.aggregation.**product_combiner**(*classifier_ensemble*, *X*)
Ensemble combination using the Product rule.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape = [n_classifiers, n_samples, n_classes]] Probabilities predicted by each base classifier in the ensemble for all samples in X.

deslib.util.aggregation.**product_rule**(*predictions*)
Apply the product fusion rule to the predicted vector of class supports (predictions).

Parameters

predictions [array of shape (n_samples, n_classifiers, n_classes)] Vector of class supports predicted by each base classifier for sample

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**weighted_majority_voting**(*classifier_ensemble*, *weights*, *X*)
Apply the weighted majority voting rule to predict the label of each sample in X. The size of the weights vector should be equal to the size of the ensemble.

Parameters

classifier_ensemble [list of shape = [n_classifiers]] Containing the ensemble of classifiers used in the aggregation scheme.

weights [array of shape (n_samples, n_classifiers)] Weights associated to each base classifier for each sample

X [array of shape (n_samples, n_features)] The input data.

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

deslib.util.aggregation.**weighted_majority_voting_rule**(*votes*, *weights*, *labels_set=None*)

Applies the weighted majority voting rule based on the votes obtained by each base classifier and their respective weights.

Parameters

votes [array of shape (n_samples, n_classifiers),] The votes obtained by each classifier for each sample.

weights [array of shape (n_samples, n_classifiers)] Weights associated to each base classifier for each sample

labels_set [(Default=None) set with the possible classes in the problem.]

Returns

predicted_label [array of shape (n_samples)] The label of each query sample predicted using the majority voting rule

Probabilistic Functions

This file contains the implementation of several functions used to estimate the competence level of a base classifiers based on posterior probabilities predicted for each class.

deslib.util.prob_functions.**ccprmod**(*supports*, *idx_correct_label*, *B=20*)

Python implementation of the ccprmod.m (Classifier competence based on probabilistic modelling) function. Matlab code is available at: http://www.mathworks.com/matlabcentral/mlc-downloads/downloads/submissions/28391/versions/6/previews/ccprmod.m/index.html

Parameters

supports: array of shape (n_samples, n_classes) Containing the supports obtained by the base classifier for each class.

idx_correct_label: array of shape (n_samples) containing the index of the correct class.

B [int (Default = 20)] number of points used in the calculation of the competence, higher values result in a more accurate estimation.

Returns

C_src [array of shape (n_samples)] representing the classifier competences at each data point

References

T.Woloszynski, M. Kurzynski, A probabilistic model of classifier competence for dynamic ensemble selection, Pattern Recognition 44 (2011) 2656–2668.

Examples

```
>>> supports = [[0.3, 0.6, 0.1],[1.0/3, 1.0/3, 1.0/3]]
>>> idx_correct_label = [1,0]
>>> ccprmod(supports,idx_correct_label)
ans = [0.784953394056843, 0.332872292262951]
```

deslib.util.prob_functions.**entropy_func**(*n_classes*, *supports*, *is_correct*)
Calculate the entropy in the support obtained by the base classifier. The value of the source competence is inverse proportional to the normalized entropy of its supports vector and the sign of competence is simply determined by the correct/incorrect classification

Parameters

n_classes [int] The number of classes in the problem

supports: array of shape (n_samples, n_classes) Containing the supports obtained by the base classifier for each class.

is_correct: array of shape (n_samples) Array with 1 whether the base classifier predicted the correct label and -1 otherwise

Returns

C_src [array of shape (n_samples)] Representing the classifier competences at each data point

References

B. Antosik, M. Kurzynski, New measures of classifier competence – heuristics and application to the design of multiple classifier systems., in: Computer recognition systems 4., 2011, pp. 197–206.

deslib.util.prob_functions.**exponential_func**(*n_classes*, *support_correct*)
Calculate the exponential function based on the support obtained by the base classifier for the correct class label.

Parameters

n_classes [int] The number of classes in the problem

support_correct: array of shape (n_samples) containing the supports obtained by the base classifier for the correct class

Returns

C_src [array of shape (n_samples)] Representing the classifier competences at each data point

deslib.util.prob_functions.**log_func**(*n_classes*, *support_correct*)
Calculate the logarithm in the support obtained by the base classifier.

Parameters

n_classes [int] The number of classes in the problem

support_correct: array of shape (n_samples) Containing the supports obtained by the base classifier for the correct class

Returns

C_src [array of shape (n_samples)] representing the classifier competences at each data point

References

T.Woloszynski, M. Kurzynski, A measure of competence based on randomized reference classifier for dynamic ensemble selection, in: International Conference on Pattern Recognition (ICPR), 2010, pp. 4194–4197.

deslib.util.prob_functions.**min_difference**(*supports*, *idx_correct_label*)
The minimum difference between the supports obtained for the correct class and the vector of class supports. The value of the source competence is negative if the sample is misclassified and positive otherwise.

Parameters

supports: array of shape (n_samples, n_classes) Containing the supports obtained by the base classifier for each class

idx_correct_label: array of shape (n_samples) Containing the index of the correct class

Returns

C_src [array of shape (n_samples)] Representing the classifier competences at each data point

References

B. Antosik, M. Kurzynski, New measures of classifier competence – heuristics and application to the design of multiple classifier systems., in: Computer recognition systems 4., 2011, pp. 197–206.

deslib.util.prob_functions.**softmax**(*w*, *theta=1.0*)

Takes an vector w of S N-element and returns a vectors where each column of the vector sums to 1, with elements exponentially proportional to the respective elements in N.

Parameters

w [array of shape = [N, M]]

theta [float (default = 1.0)] used as a multiplier prior to exponentiation.

Returns

dist [array of shape = [N, M]] Which the sum of each row sums to 1 and the elements are exponentially proportional to the respective elements in N

Instance Hardness

This file contains the implementation of different measures of instance hardness.

deslib.util.instance_hardness.**hardness_region_competence**(*neighbors_idx*, *labels*, *safe_k*)

Calculate the Instance hardness of the sample based on its neighborhood. The sample is deemed hard to classify when there is overlap between different classes in the region of competence. This method does not takes into account the target label of the test sample

This hardness measure is used to select whether use DS or use the KNN for the classification of a given query sample

Parameters

neighbors_idx [array of shape = [n_samples_test, k]] Indices of the nearest neighbors for each considered sample

labels [array of shape = [n_samples_train]] labels associated with each training sample

safe_k [int] Number of neighbors used to estimate the hardness of the corresponding region

Returns

hardness [array of shape = [n_samples_test]] The Hardness level associated with each example.

References

Smith, M.R., Martinez, T. and Giraud-Carrier, C., 2014. An instance level analysis of data complexity. Machine learning, 95(2), pp.225-256

deslib.util.instance_hardness.**kdn_score**(*X*, *y*, *k*)

Calculates the K-Disagreeing Neighbors score (KDN) of each sample in the input dataset.

Parameters

X [array of shape (n_samples, n_features)] The input data.

y [array of shape (n_samples)] class labels of each example in X.

k [int] Neighborhood size for calculating the KDN score.

Returns

score [array of shape = [n_samples,1]] KDN score of each sample in X.

neighbors [array of shape = [n_samples,k]] Indexes of the k neighbors of each sample in X.

References

M. R. Smith, T. Martinez, C. Giraud-Carrier, An instance level analysis of data complexity, Machine Learning 95 (2) (2014) 225-256.

Frienemy Pruning

Implementation of the Dynamic Frienemy Pruning (DFP) algorithm for online pruning of base classifiers.

References

Oliveira, D.V.R., Cavalcanti, G.D.C. and Sabourin, R., Online Pruning of Base Classifiers for Dynamic Ensemble Selection, Pattern Recognition, vol. 72, December 2017, pp 44-58.

Cruz, Rafael MO, Dayvid VR Oliveira, George DC Cavalcanti, and Robert Sabourin. "FIRE-DES++: Enhanced online pruning of base classifiers for dynamic ensemble selection." Pattern Recognition 85 (2019): 149-160.

deslib.util.dfp.**frienemy_pruning**(*X_query*, *X_dsel*, *y_dsel*, *ensemble*, *k*)

Implements the Online Pruning method (frienemy) which prunes base classifiers that do not cross the region of competence of a given instance. A classifier crosses the region of competence if it correctly classify at least one sample for each different class in the region.

Parameters

X_query [array-like of shape (n_samples, n_features)] Test set.

X_dsel [array-like of shape (n_samples, n_features)] Dynamic selection set.

y_dsel [array-like of shape (n_samples,)] The target values (Dynamic selection set).

ensemble [list of shape = [n_classifiers]] The ensemble of classifiers to be pruned.

k [int] Number of neighbors used to compute the regions of competence.

Returns

DFP_mask [array-like of shape = [n_samples, n_classifiers]] Mask containing 1 for the selected base classifier and 0 otherwise.

deslib.util.dfp.**frienemy_pruning_preprocessed**(*neighbors*, *y_val*, *hit_miss*)

Implements the Online Pruning method (frienemy) which prunes base classifiers that do not cross the region of competence of a given instance. A classifier crosses the region of competence if it correctly classify at least one sample for each different class in the region.

Parameters

neighbors [array-like of shape (n_samples, n_neighbors)] Indices of the k nearest neighbors.

y_val [array-like of shape (n_samples,)] The target values (class labels).

hit_miss [array-like of shape (n_samples, n_classifiers)] Matrix containing 1 when the base classifier made the correct prediction, 0 otherwise.

Returns

DFP_mask [array-like of shape = [n_samples, n_classifiers]] Mask containing 1 for the selected base classifier and 0 otherwise.

Notes

This implementation assumes the regions of competence of each query example (neighbors) and the predictions for the dynamic selection data (hit_miss) were already pre-computed.

KNN-Equality

class deslib.util.knne.**KNNE**(*n_neighbors=7*, *knn_classifier='sklearn'*, ***kwargs*)

" Implementation of the K-Nearest Neighbors-Equality technique.

This implementation fits a different KNN method for each class, and search on each class for the nearest examples.

Parameters

n_neighbors [int, (default = 7)] Number of neighbors to use by default for *kneighbors()* queries.

algorithm [str = ['knn', 'faiss]', (default = 'knn')] Whether to use scikit-learn or faiss for nearest neighbors estimation.

References

Sierra, Basilio, Elena Lazkano, Itziar Irigoien, Ekaitz Jauregi, and Iñigo Mendialdua. "K nearest neighbor equality: giving equal chance to all existing classes." Information Sciences 181, no. 23 (2011): 5158-5168.

Mendialdua, Iñigo, José María Martínez-Otzeta, I. Rodriguez-Rodriguez, T. Ruiz-Vazquez, and Basilio Sierra. "Dynamic selection of the best base classifier in one versus one." Knowledge-Based Systems 85 (2015): 298-306.

Cruz, Rafael MO, Dayvid VR Oliveira, George DC Cavalcanti, and Robert Sabourin. "FIRE-DES++: Enhanced online pruning of base classifiers for dynamic ensemble selection." Pattern Recognition 85 (2019): 149-160.

fit(*X*, *y*)

Fit the model according to the given training data.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

kneighbors(*X=None*, *n_neighbors=None*, *return_distance=True*)

Finds the K-neighbors of a point. Returns indices of and distances to the neighbors of each point.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] The query point or points. If not provided, neighbors of each indexed point are returned. In this case, the query point is not considered its own neighbor.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dist [array] Array representing the lengths to points, only present if return_distance=True

ind [array] Indices of the nearest points in the population matrix.

predict(*X*)

Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

preds [array, shape (n_samples,)] Class labels for samples in X.

predict_proba(*X*)

Return probability estimates for the test data X.

Parameters

X [array-like, shape (n_query, n_features), or (n_query, n_indexed) if metric == 'precomputed'] Test samples.

Returns

proba [array of shape (n_samples, n_classes), or a list of n_outputs] of such arrays if n_outputs > 1. The class probabilities of the input samples. Classes are ordered by lexicographic order.

FAISS Wrapper

class deslib.util.faiss_knn_wrapper.**FaissKNNClassifier**(*n_neighbors=5*, *n_jobs=None*, *algorithm='brute'*, *n_cells=100*, *n_probes=1*)

Scikit-learn wrapper interface for Faiss KNN.

Parameters

n_neighbors [int (Default = 5)] Number of neighbors used in the nearest neighbor search.

n_jobs [int (Default = None)]

The number of jobs to run in parallel for both fit and predict. If -1, then the number of jobs is set to the number of cores.

algorithm [{'brute', 'voronoi'} (Default = 'brute')] Algorithm used to compute the nearest neighbors:

- 'brute' will use the :class: *IndexFlatL2* class from faiss.
- 'voronoi' will use `IndexIVFFlat` class from faiss.

- 'hierarchical' will use `IndexHNSWFlat` class from faiss.

Note that selecting 'voronoi' the system takes more time during training, however it can significantly improve the search time on inference. 'hierarchical' produce very fast and accurate indexes, however it has a higher memory requirement. It's recommended when you have a lots of RAM or the dataset is small.

For more information see: https://github.com/facebookresearch/faiss/wiki/Guidelines-to-choose-an-index

n_cells [int (Default = 100)] Number of voronoi cells. Only used when algorithm=='voronoi'.

n_probes [int (Default = 1)] Number of cells that are visited to perform the search. Note that the search time roughly increases linearly with the number of probes. Only used when algorithm=='voronoi'.

References

Johnson Jeff, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with gpus." arXiv preprint arXiv:1702.08734 (2017).

fit (*X*, *y*)
Fit the model according to the given training data.

Parameters

X [array of shape (n_samples, n_features)] Data used to fit the model.

y [array of shape (n_samples)] class labels of each example in X.

kneighbors (*X*, *n_neighbors=None*, *return_distance=True*)
Finds the K-neighbors of a point.

Parameters

X [array of shape (n_samples, n_features)] The input data.

n_neighbors [int] Number of neighbors to get (default is the value passed to the constructor).

return_distance [boolean, optional. Defaults to True.] If False, distances will not be returned

Returns

dists [list of shape = [n_samples, k]] The distances between the query and each sample in the region of competence. The vector is ordered in an ascending fashion.

idx [list of shape = [n_samples, k]] Indices of the instances belonging to the region of competence of the given query sample.

predict (*X*)
Predict the class label for each sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

preds [array, shape (n_samples,)] Class labels for samples in X.

predict_proba (*X*)
Estimates the posterior probabilities for sample in X.

Parameters

X [array of shape (n_samples, n_features)] The input data.

Returns

preds_proba [array of shape (n_samples, n_classes)] Probabilities estimates for each sample in X.

Datasets

This file contains routines to generate 2D classification datasets that can be used to test the performance of different machine learning algorithms.

- P2 Dataset
- Circle and Square
- Banana
- Banana 2

deslib.util.datasets.**make_P2**(*size_classes*, *random_state=None*)

Generate the P2 Dataset:

The P2 is a two-class problem, presented by Valentini[1], in which each class is defined in multiple decision regions delimited by polynomial and trigonometric functions (E1, E2, E3 and E4):

$$to$$

$$E1(x) = sin(x) + 5$$
$$E2(x) = (x-2)^2 + 1$$
$$E3(x) = -0.1 \cdot x^2 + 0.6sin(4x) + 8$$
$$E4(x) = \frac{(x-10)^2}{2} + 7.902 (3.1)$$

Parameters

size_classes [list with the number of samples for each class.]

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Returns

X [array of shape = [size_classes, 2]] The generated data points.

y [array of shape = [size_classes]] Class labels associated with each class.

References

G. Valentini, An experimental bias-variance analysis of svm ensembles based on resampling techniques, IEEE Transactions on Systems, Man, and Cybernetics, Part B 35 (2005) 1252–1271.

deslib.util.datasets.**make_banana**(*size_classes*, *na=0.1*, *random_state=None*)

Generate the Banana dataset.

Parameters

size_classes [list with the number of samples for each class.]

na [float (Default = 0.2),] Noise amplitude. It must be < 1.0

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Returns

X [array of shape = [size_classes, 2]] The generated data points.

y [array of shape = [size_classes]] Class labels associated with each class.

References

Kuncheva, Ludmila I. Combining pattern classifiers: methods and algorithms. John Wiley & Sons, 2004.

deslib.util.datasets.**make_banana2**(*size_classes*, *sigma=1*, *random_state=None*)

Generate the Banana dataset similar to the Matlab PRTools toolbox.

Parameters

size_classes [list with the number of samples for each class.]

sigma [float (Default = 1),] variance of the normal distribution

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Returns

X [array of shape = [size_classes, 2]] The generated data points.

y [array of shape = [size_classes]] Class labels associated with each class.

References

R.P.W. Duin, P. Juszczak, D.de Ridder, P. Paclik, E. Pekalska, D.M.Tax, Prtools, a matlab toolbox for pattern recognition, 2004. URL ⟨http://www.prtools.org⟩.

deslib.util.datasets.**make_circle_square**(*size_classes*, *random_state=None*)

Generate the circle square dataset.

Parameters

size_classes [list with the number of samples for each class.]

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Returns

X [array of shape = [size_classes, 2]] The generated data points.

y [array of shape = [size_classes]] Class labels associated with each class.

References

P. Henniges, E. Granger, R. Sabourin, Factors of overtraining with fuzzy artmap neural networks, International Joint Conference on Neural Networks (2005) 1075–1080.

`deslib.util.datasets.`**`make_xor`**(*n_samples*, *random_state=None*)

Generate the exclusive-or (XOR) dataset.

Parameters

n_samples [int] Number of generated data points.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

Returns

X [array of shape = [size_classes, 2]] The generated data points.

y [array of shape = [size_classes]] Class labels associated with each class.

3.3 General examples

Examples showing how to use different aspect of the library

3.3.1 Simple example

In this example we show how to apply different DCS and DES techniques for a classification dataset.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from deslib.des import METADES
from deslib.des import KNORAE


# Setting up the random state to have consistent results
rng = np.random.RandomState(42)

# Generate a classification dataset
X, y = make_classification(n_samples=1000, random_state=rng)
# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
```

(continues on next page)

(continued from previous page)

```
                                                    random_state=rng)

# Split the data into training and DSEL for DS techniques
X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train,
                                                    test_size=0.5,
                                                    random_state=rng)

# Initialize the DS techniques. DS methods can be initialized without
# specifying a single input parameter. In this example, we just pass the random
# state in order to always have the same result.
kne = KNORAE(random_state=rng)
meta = METADES(random_state=rng)

# Fitting the des techniques
kne.fit(X_dsel, y_dsel)
meta.fit(X_dsel, y_dsel)

# Calculate classification accuracy of each technique
print('Evaluating DS techniques:')
print('Classification accuracy KNORA-Eliminate: ',
      kne.score(X_test, y_test))
print('Classification accuracy META-DES: ', meta.score(X_test, y_test))
```

Total running time of the script: (0 minutes 0.000 seconds)

3.3.2 Measuring the influence of the region of competence

This example shows how the size of the region of competence (parameter k) can influence the final performance of DS techniques.

In this example we vary the value of the parameter k from 3 to 15 and measure the performance of 7 different dynamic selection technique using the same pool of classifiers.

Let's start by importing all required modules. In this example we use the new sklearn-OpenML interface to fetch the diabetes classification problem.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from deslib.dcs import LCA
# DCS techniques
from deslib.dcs import MCB
from deslib.dcs import OLA
from deslib.dcs import Rank
# DES techniques
from deslib.des import DESP
from deslib.des import KNORAE
from deslib.des import KNORAU

rng = np.random.RandomState(123456)
```

(continues on next page)

(continued from previous page)

```
data = fetch_openml(name='diabetes', cache=False, as_frame=False)
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rng)

# Normalizing the dataset to have 0 mean and unit variance.
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

pool_classifiers = BaggingClassifier(Perceptron(max_iter=100),
                                     random_state=rng)
pool_classifiers.fit(X_train, y_train)

# Setting with_IH
mcb = MCB(pool_classifiers, random_state=rng)
ola = OLA(pool_classifiers)
des_p = DESP(pool_classifiers)
knu = KNORAU(pool_classifiers)
lca = LCA(pool_classifiers)
kne = KNORAE(pool_classifiers)
rank = Rank(pool_classifiers)
list_ds_methods = [mcb, ola, des_p, knu, lca, kne, rank]
names = ['MCB', 'OLA', 'DES-P', 'KNORA-U', 'LCA', 'KNORA-E', 'Rank']

k_value_list = range(3, 16)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/deslib/envs/v0.3.5/lib/python3.6/
↪site-packages/sklearn/datasets/_openml.py:407: UserWarning: Multiple active␣
↪versions of the dataset matching the name diabetes exist. Versions may be␣
↪fundamentally different, returning version 1.
  " {version}.".format(name=name, version=res[0]['version']))
```

Plot accuracy x region of competence size.

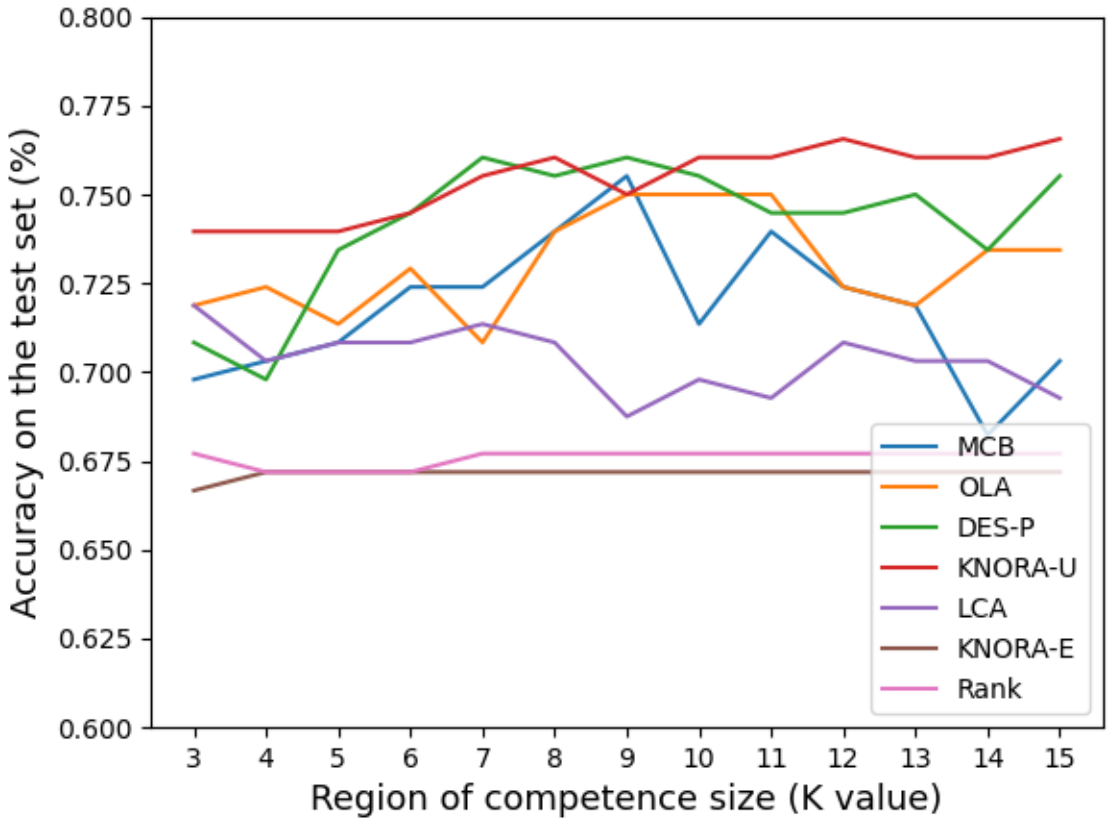
We can see the this parameter can have a huge influence in the performance of certain DS techniques. The main exception being the KNORA-E and Rank which have built-in mechanism to automatically adjust the region of competence size during the competence level estimation.

```
fig, ax = plt.subplots()
for ds_method, name in zip(list_ds_methods, names):
    accuracy = []
    for k in k_value_list:
        ds_method.k = k
        ds_method.fit(X_train, y_train)
        accuracy.append(ds_method.score(X_test, y_test))
    ax.plot(k_value_list, accuracy, label=name)

plt.xticks(k_value_list)
ax.set_ylim(0.60, 0.80)
ax.set_xlabel('Region of competence size (K value)', fontsize=13)
ax.set_ylabel('Accuracy on the test set (%)', fontsize=13)
```

(continues on next page)

(continued from previous page)

```
ax.legend(loc='lower right')
plt.show()
```



Total running time of the script: (0 minutes 17.253 seconds)

3.3.3 Dynamic selection vs K-NN: Using instance hardness

One aspect about dynamic selection techniques is that it can better deal with the classification of test examples associated with high degree of instance hardness. Such examples are often found close to the border of the classes, with the majority of its neighbors belonging to different classes. On the other hand, the KNN method, which is often used to estimate the region of competence in DS methods works better in the classification of examples associated with low instance hardness [1].

DESlib already implements a switch mechanism between DS techniques and the KNN classifier according to the hardness level of an instance. This example varies the threshold in which KNN is used for classification instead of DS methods. It also compares the classification results with the standard KNN as a baseline.

The switch mechanism also reduces the computational cost involved since only part of the test samples are classified by the DS method.

References

[1] Cruz, Rafael MO, et al. "Dynamic Ensemble Selection VS K-NN: why and when Dynamic Selection obtains

higher classification performance?." arXiv preprint arXiv:1804.07882 (2018).

Let's start by importing all required modules. In this example we use the new sklearn-OpenML interface to fetch the diabetes classification problem.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.tree import DecisionTreeClassifier

from deslib.dcs import MCB
from deslib.dcs import OLA
from deslib.dcs import Rank
from deslib.des import DESP
from deslib.des import KNORAE
from deslib.des import KNORAU

rng = np.random.RandomState(123456)

data = fetch_openml(name='diabetes', cache=False, as_frame=False)
X = data.data
y = data.target
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=rng)

# Normalizing the dataset to have 0 mean and unit variance.
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Training a pool of classifiers using the bagging technique.
pool_classifiers = BaggingClassifier(DecisionTreeClassifier(random_state=rng),
                                     random_state=rng)
pool_classifiers.fit(X_train, y_train)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/deslib/envs/v0.3.5/lib/python3.6/
↪site-packages/sklearn/datasets/_openml.py:407: UserWarning: Multiple active␣
↪versions of the dataset matching the name diabetes exist. Versions may be␣
↪fundamentally different, returning version 1.
  " {version}.".format(name=name, version=res[0]['version']))

BaggingClassifier(base_estimator=DecisionTreeClassifier(random_
↪state=RandomState(MT19937) at 0x7F5C0FDAA570),
                  random_state=RandomState(MT19937) at 0x7F5C0FDAA570)
```

Setting DS method to use the switch mechanism

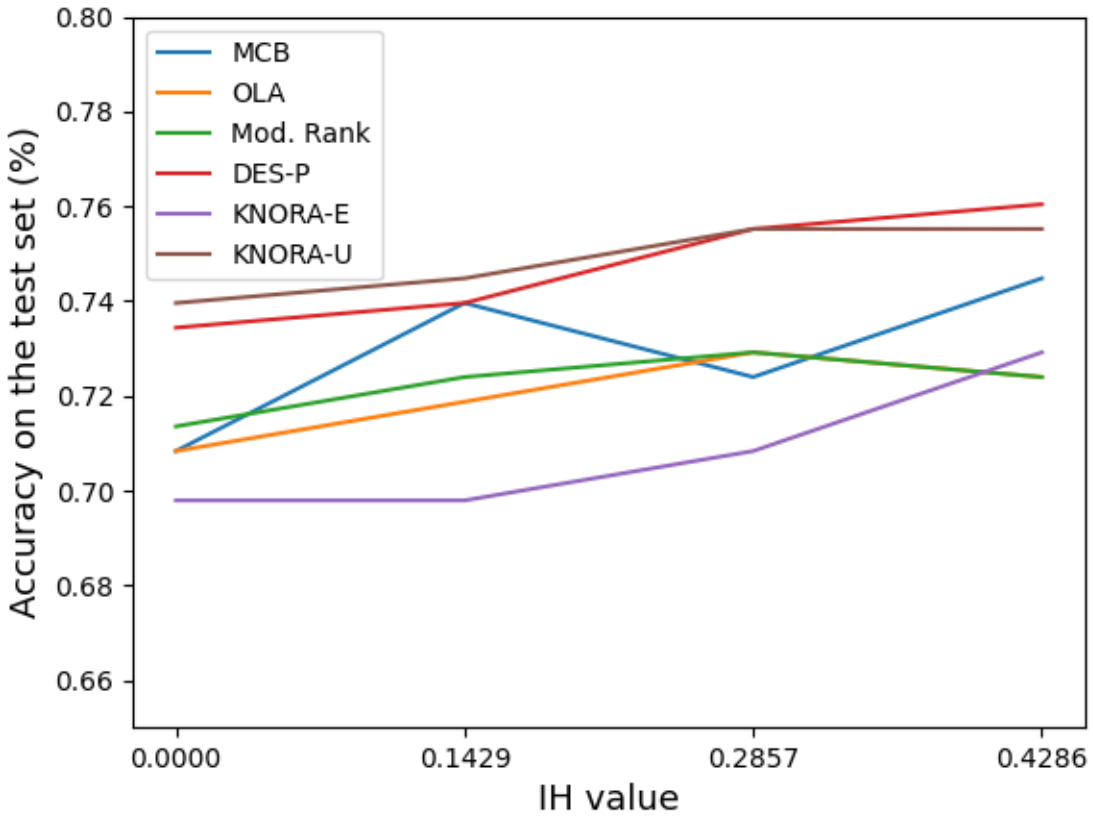
In order to activate the functionality to switch between DS and KNN according to the instance hardness level we need to set the DS techniques to use this information. This is done by setting the hyperparameter *with_IH* to True. In this example we consider four different values for te threshold

```
mcb = MCB(pool_classifiers, with_IH=True, random_state=rng)
ola = OLA(pool_classifiers, with_IH=True, random_state=rng)
```

(continues on next page)

(continued from previous page)

```
rank = Rank(pool_classifiers, with_IH=True, random_state=rng)
des_p = DESP(pool_classifiers, with_IH=True, random_state=rng)
kne = KNORAE(pool_classifiers, with_IH=True, random_state=rng)
knu = KNORAU(pool_classifiers, with_IH=True, random_state=rng)
list_ih_values = [0.0, 1./7., 2./7., 3./7.]

list_ds_methods = [method.fit(X_train, y_train) for method in
                   [mcb, ola, rank, des_p, kne, knu]]
names = ['MCB', 'OLA', 'Mod. Rank', 'DES-P', 'KNORA-E', 'KNORA-U']

# Plot accuracy x IH
fig, ax = plt.subplots()
for ds_method, name in zip(list_ds_methods, names):
    accuracy = []
    for idx_ih, ih_rate in enumerate([0.0, 0.14, 0.28, 0.42]):
        ds_method.IH_rate = ih_rate
        accuracy.append(ds_method.score(X_test, y_test))
    ax.plot(list_ih_values, accuracy, label=name)

plt.xticks(list_ih_values)
ax.set_ylim(0.65, 0.80)
ax.set_xlabel('IH value', fontsize=13)
ax.set_ylabel('Accuracy on the test set (%)', fontsize=13)
ax.legend()

plt.show()
```



Total running time of the script: (0 minutes 9.403 seconds)

3.3.4 Calibrating base classifiers to estimate probabilities

In this example we show how to apply different DCS and DES techniques for a classification dataset.

A very important aspect in dynamic selection is the generation of a pool of classifiers. A common practice in the dynamic selection literature is to use the Bagging (Bootstrap Aggregating) method to generate a pool containing base classifiers that are both diverse and informative.

In this example we generate a pool of classifiers using the Bagging technique implemented on the Scikit-learn library. Then, we compare the results obtained by combining this pool of classifiers using the standard Bagging combination approach versus the application of dynamic selection technique to select the set of most competent classifiers

```
import numpy as np
from sklearn.calibration import CalibratedClassifierCV
from sklearn.datasets import load_breast_cancer
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from deslib.dcs.a_priori import APriori
from deslib.dcs.mcb import MCB
from deslib.dcs.ola import OLA
```

(continues on next page)

(continued from previous page)

```
from deslib.des.des_p import DESP
from deslib.des.knora_e import KNORAE
from deslib.des.knora_u import KNORAU
from deslib.des.meta_des import METADES
```

Preparing the dataset

In this part we load the breast cancer dataset from scikit-learn and preprocess it in order to pass to the DS models. An important point here is to normalize the data so that it has zero mean and unit variance, which is a common requirement for many machine learning algorithms. This step can be easily done using the StandardScaler class.

```
rng = np.random.RandomState(123)
data = load_breast_cancer()
X = data.data
y = data.target
# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
                                                    random_state=rng)

# Scale the variables to have 0 mean and unit variance
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Split the data into training and DSEL for DS techniques
X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train,
                                                    test_size=0.5,
                                                    random_state=rng)

# Train a pool of 100 base classifiers
pool_classifiers = BaggingClassifier(Perceptron(max_iter=10),
                                     n_estimators=100, random_state=rng)
pool_classifiers.fit(X_train, y_train)

# Initialize the DS techniques
knorau = KNORAU(pool_classifiers)
kne = KNORAE(pool_classifiers)
desp = DESP(pool_classifiers)
ola = OLA(pool_classifiers)
mcb = MCB(pool_classifiers, random_state=rng)
```

Calibrating base classifiers

Some dynamic selection techniques requires that the base classifiers estimate probabilities in order to estimate its competence level. Since the Perceptron model is not a probabilistic classifier (does not implements the predict_proba method, it needs to be calibrated for probability estimation before being used by such DS techniques. This step can be conducted using the CalibrateClassifierCV class from scikit-learn. Note that in this example we pass a prefited pool of classifiers to the calibration method in order to use exactly the same pool used in the other DS methods.

```
calibrated_pool = []
for clf in pool_classifiers:
    calibrated = CalibratedClassifierCV(base_estimator=clf, cv='prefit')
    calibrated.fit(X_dsel, y_dsel)
```

(continues on next page)

(continued from previous page)

```
    calibrated_pool.append(calibrated)

apriori = APriori(calibrated_pool, random_state=rng)
meta = METADES(calibrated_pool)


knorau.fit(X_dsel, y_dsel)
kne.fit(X_dsel, y_dsel)
desp.fit(X_dsel, y_dsel)
ola.fit(X_dsel, y_dsel)
mcb.fit(X_dsel, y_dsel)
apriori.fit(X_dsel, y_dsel)
meta.fit(X_dsel, y_dsel)
```

Evaluating the methods

Let's now evaluate the methods on the test set. We also use the performance of Bagging (pool of classifiers without any selection) as a baseline comparison. We can see that the majority of DS methods achieve higher classification accuracy.

```
print('Evaluating DS techniques:')
print('Classification accuracy KNORA-Union: ',
      knorau.score(X_test, y_test))
print('Classification accuracy KNORA-Eliminate: ',
      kne.score(X_test, y_test))
print('Classification accuracy DESP: ', desp.score(X_test, y_test))
print('Classification accuracy OLA: ', ola.score(X_test, y_test))
print('Classification accuracy A priori: ', apriori.score(X_test, y_test))
print('Classification accuracy MCB: ', mcb.score(X_test, y_test))
print('Classification accuracy META-DES: ', meta.score(X_test, y_test))
print('Classification accuracy Bagging: ',
      pool_classifiers.score(X_test, y_test))
```

Total running time of the script: (0 minutes 0.000 seconds)

3.3.5 Using the Dynamic Frienemy Pruning (DFP)

In this example we show how to apply the dynamic frienemy pruning (DFP) to different dynamic selection techniques.

The DFP method is an online pruning model which analyzes the region of competence to know if it is composed of samples from different classes (indecision region). Then, it remove the base classifiers that do not correctly classifies at least a pair of samples coming from different classes, i.e., the base classifiers that cannot separate the classes in the local region. More information on this method can be found in refs [1] and [2].

DES techniques using the DFP algorithm are called FIRE-DES (Frienemy Indecision REgion Dynamic Ensemble Selection). The FIRE-DES is shown to significantly improve the performance of several dynamic selection algorithms when dealing with imbalanced classification problems as it avoids the classifiers that are biased towards the majority class in predicting the label for the query.

References

[1] Oliveira, D.V.R., Cavalcanti, G.D.C. and Sabourin, R., "Online Pruning of Base Classifiers for Dynamic Ensemble Selection", Pattern Recognition, vol. 72, 2017, pp 44-58.

[2] Cruz, R.M.O., Oliveira, D.V.R., Cavalcanti, G.D.C. and Sabourin, R., "FIRE-DES++: Enhanced online pruning of base classifiers for dynamic ensemble selection"., Pattern Recognition, vol. 85, 2019, pp 149-160.

```
import numpy as np
from sklearn.datasets import make_classification
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
import matplotlib.pyplot as plt
from deslib.dcs import APosteriori
from deslib.dcs import APriori
from deslib.dcs import LCA
from deslib.dcs import OLA
from deslib.des import DESP
from deslib.des import METADES

rng = np.random.RandomState(654321)

# Generate an imbalanced classification dataset
X, y = make_classification(n_classes=2, n_samples=2000, weights=[0.05, 0.95],
                           random_state=rng)
# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
                                                    random_state=rng)

# Split the data into training and DSEL for DS techniques
X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train,
                                                    test_size=0.5,
                                                    random_state=rng)
# Considering a pool composed of 10 base classifiers
pool_classifiers = RandomForestClassifier(n_estimators=10, random_state=rng,
                                          max_depth=10)
pool_classifiers.fit(X_train, y_train)

ds_names = ['A Priori', 'A Posteriori', 'OLA', 'LCA', 'DES-P', 'META-DES']

# DS techniques without DFP
apriori = APriori(pool_classifiers, random_state=rng)
aposteriori = APosteriori(pool_classifiers, random_state=rng)
ola = OLA(pool_classifiers)
lca = LCA(pool_classifiers)
desp = DESP(pool_classifiers)
meta = METADES(pool_classifiers)

# FIRE-DS techniques (with DFP)
fire_apriori = APriori(pool_classifiers, DFP=True, random_state=rng)
fire_aposteriori = APosteriori(pool_classifiers, DFP=True, random_state=rng)
fire_ola = OLA(pool_classifiers, DFP=True)
fire_lca = LCA(pool_classifiers, DFP=True)
fire_desp = DESP(pool_classifiers, DFP=True)
fire_meta = METADES(pool_classifiers, DFP=True)

list_ds = [apriori, aposteriori, ola, lca, desp, meta]
list_fire_ds = [fire_apriori, fire_aposteriori, fire_ola,
                fire_lca, fire_desp, fire_meta]

scores_ds = []
for ds in list_ds:
```

(continues on next page)

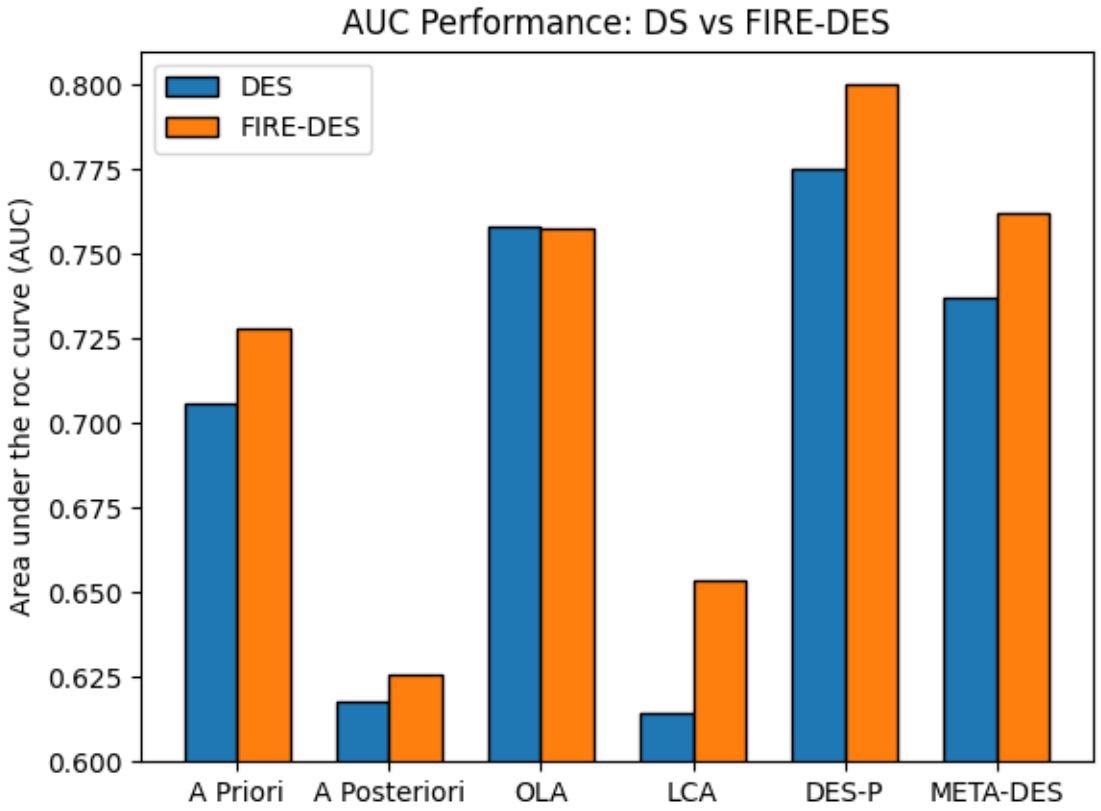
(continued from previous page)

```
    ds.fit(X_dsel, y_dsel)
    scores_ds.append(roc_auc_score(y_test, ds.predict(X_test)))

scores_fire_ds = []
for fire_ds in list_fire_ds:
    fire_ds.fit(X_dsel, y_dsel)
    scores_fire_ds.append(roc_auc_score(y_test, fire_ds.predict(X_test)))
```

Comparing DS techniques with FIRE-DES techniques

Let's now evaluate the DES methods on the test set. Since we are dealing with imbalanced data, we use the area under the roc curve (AUC) as performance metric instead of classification accuracy. The AUC can be easily calculated using the *sklearn.metrics.roc_auc_score* function from scikit-learn.

```
width = 0.35
ind = np.arange(len(ds_names))
plt.bar(ind, scores_ds, width, label='DES', edgecolor='k')
plt.bar(ind + width, scores_fire_ds, width, label='FIRE-DES', edgecolor='k')

plt.ylabel('Area under the roc curve (AUC)')
plt.title('AUC Performance: DS vs FIRE-DES')
plt.ylim((0.60, 0.81))
plt.xticks(ind + width / 2, ds_names)
plt.legend(loc='best')
plt.show()
```



Total running time of the script: (0 minutes 2.298 seconds)

3.3.6 Comparing dynamic selection with baseline static methods

In this example we compare the performance of DS techinques with the static ensemble methods. DESlib offer the implementation of static ensemble methods in the *deslib.static* module. The following techniques are considered:

Static methods used as baseline comparison are in the *deslib.static* module. They are:

Majority Voting: The outputs of all base classifiers in the pool are combined using the majority voting rule

Static Selection: A fraction of the best performing classifiers (based on the validation data, is selected to compose the ensemble).

Single Best: The base classifier with the highest classification accuracy in the validation set is selected for classification

Stacked classifier: The outputs of all base classifiers are passed down to a meta-estimator which combines the . The meta-estimator is trained based on the outputs of the base classifiers on the training data.

These techniques are used in the dynamic selection literature as a baseline comparison (for more information see references [1] and [2])

At the end we also present the result of the **Oracle**, which is an abastract model which always selects the base classifier that predicted the correct label if such classifier exists. From the dynamic selection point of view, the Oracle is seen as the upper limit performance that can be achieved with the given pool of classifiers.

References

[1] Britto, Alceu S., Robert Sabourin, and Luiz ES Oliveira. "Dynamic selection of classifiers—a comprehensive review." Pattern Recognition 47.11 (2014): 3665-3680.

[2] R. M. O. Cruz, R. Sabourin, and G. D. Cavalcanti, "Dynamic classifier selection: Recent advances and perspectives," Information Fusion, vol. 41, pp. 195 – 216, 2018.

[3] Kuncheva, Ludmila I. "A theoretical study on six classifier fusion strategies." IEEE Transactions on Pattern Analysis & Machine Intelligence, (2002): 281-286.

```
import matplotlib.pyplot as plt
from matplotlib.ticker import FuncFormatter
from matplotlib.cm import get_cmap
import numpy as np

# Example of a dcs techniques
from deslib.dcs import OLA
from deslib.dcs import MCB
from deslib.des import DESP
from deslib.des import KNORAU
from deslib.des.knora_e import KNORAE
from deslib.des import KNOP
from deslib.des import METADES

from sklearn.datasets import make_classification
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
# Example of a des techniques

# Example of stacked model
from deslib.static import (StackedClassifier,
                           SingleBest,
                           StaticSelection,
                           Oracle)


rng = np.random.RandomState(123)

# Generate a classification dataset
X, y = make_classification(n_samples=2000,
                           n_classes=3,
                           n_informative=6,
                           random_state=rng)

# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
                                                    random_state=rng)

X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train,
                                                    test_size=0.50,
                                                    random_state=rng)

pool_classifiers = BaggingClassifier(base_estimator=DecisionTreeClassifier(),
                                     n_estimators=100,
                                     random_state=rng)
pool_classifiers.fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
# Setting up static methods.
stacked = StackedClassifier(pool_classifiers)
static_selection = StaticSelection(pool_classifiers)
single_best = SingleBest(pool_classifiers)

# Initialize a DS technique. Here we specify the size of
# the region of competence (5 neighbors)
knorau = KNORAU(pool_classifiers, random_state=rng)
kne = KNORAE(pool_classifiers, random_state=rng)
desp = DESP(pool_classifiers, random_state=rng)
ola = OLA(pool_classifiers, random_state=rng)
mcb = MCB(pool_classifiers, random_state=rng)
knop = KNOP(pool_classifiers, random_state=rng)
meta = METADES(pool_classifiers, random_state=rng)

names = ['Single Best', 'Static Selection', 'Stacked',
         'KNORA-U', 'KNORA-E', 'DES-P', 'OLA', 'MCB', 'KNOP', 'META-DES']

methods = [single_best, static_selection, stacked,
           knorau, kne, desp, ola, mcb, knop, meta]

# Fit the DS techniques
scores = []
for method, name in zip(methods, names):
    method.fit(X_dsel, y_dsel)
    scores.append(method.score(X_test, y_test))
    print("Classification accuracy {} = {}"
          .format(name, method.score(X_test, y_test)))
```

Out:

```
Classification accuracy Single Best = 0.774
Classification accuracy Static Selection = 0.834
Classification accuracy Stacked = 0.804
Classification accuracy KNORA-U = 0.838
Classification accuracy KNORA-E = 0.83
Classification accuracy DES-P = 0.838
Classification accuracy OLA = 0.802
Classification accuracy MCB = 0.83
Classification accuracy KNOP = 0.84
Classification accuracy META-DES = 0.862
```

Plotting the results

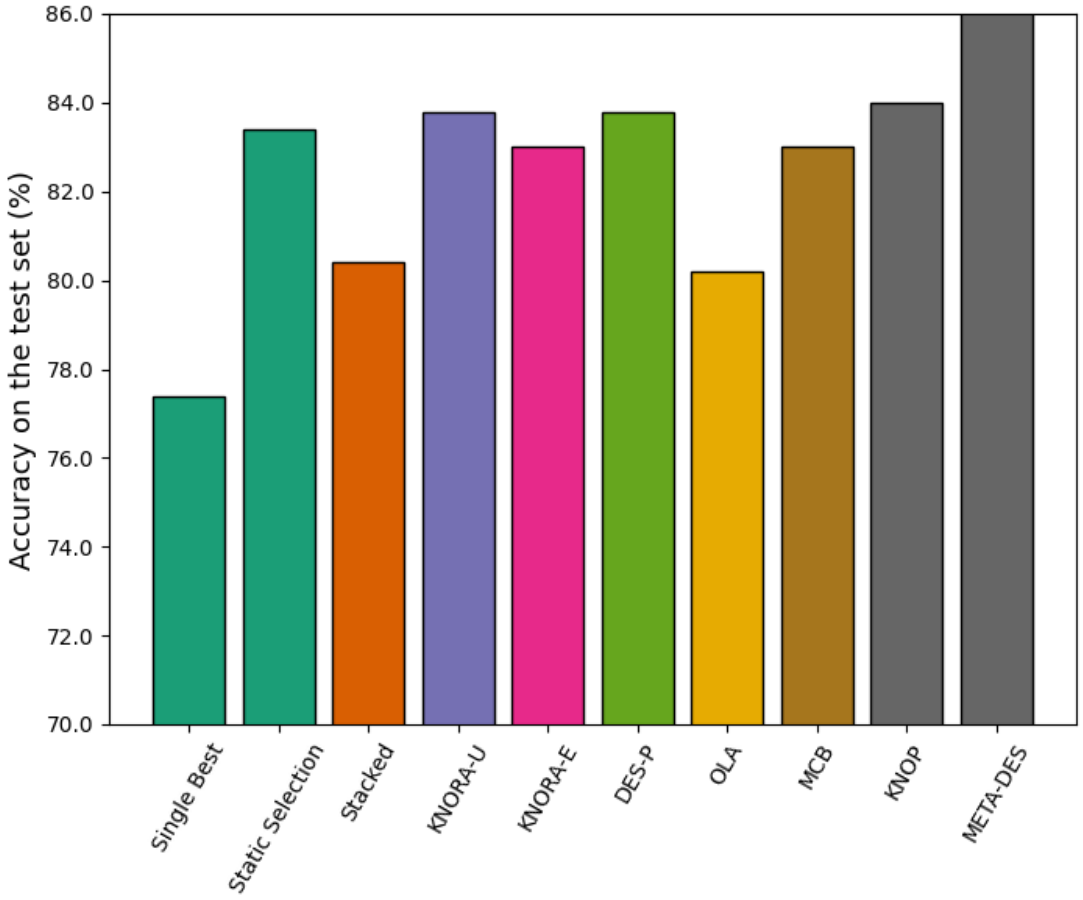
Let's now evaluate the methods on the test set.

```
cmap = get_cmap('Dark2')
colors = [cmap(i) for i in np.linspace(0, 1, 10)]
fig, ax = plt.subplots(figsize=(8, 6.5))
pct_formatter = FuncFormatter(lambda x, pos: '{:.1f}'.format(x * 100))
ax.bar(np.arange(len(methods)),
       scores,
       color=colors,
       tick_label=names,
```

(continues on next page)

(continued from previous page)

```
        edgecolor='k')

ax.set_ylim(0.70, 0.86)
ax.set_ylabel('Accuracy on the test set (%)', fontsize=13)
ax.yaxis.set_major_formatter(pct_formatter)
for tick in ax.get_xticklabels():
    tick.set_rotation(60)
plt.subplots_adjust(bottom=0.18)

plt.show()
```



The Oracle results

OracleAbstract method that always selects the base classifier that predicts the correct label if such classifier exists. This method is often used to measure the upper-limit performance that can be achieved by a dynamic classifier selection technique. It is used as a benchmark by several dynamic selection algorithms. We can see the Oracle performance is close to 100%, which is an almost 15% gap to the best performing method.

```
oracle = Oracle(pool_classifiers).fit(X_train, y_train)
print('Oracle result: {}' .format(oracle.score(X_test, y_test)))
```

Out:

```
Oracle result: 0.998
```

Total running time of the script: (0 minutes 5.528 seconds)

3.3.7 Comparing dynamic selection with Random Forest

In this example we use a pool of classifiers generated using the Random Forest method rather than Bagging. We also show how to change the size of the region of competence, used to estimate the local competence of the base classifiers.

This demonstrates that the library accepts any kind of base classifiers as long as they implement the predict and predict proba functions. Moreover, any ensemble generation method such as Boosting or Rotation Trees can be used to generate a pool containing diverse base classifiers. We also included the performance of the RandomForest classifier as a baseline comparison.

```
import matplotlib.pyplot as plt
import numpy as np
from matplotlib.cm import get_cmap
from matplotlib.ticker import FuncFormatter
from sklearn.datasets import fetch_openml
# Pool of base classifiers
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

from deslib.dcs.mcb import MCB
# Example of a dcs techniques
from deslib.dcs.ola import OLA
# Example of a des techniques
from deslib.des.des_p import DESP
from deslib.des.knora_e import KNORAE
from deslib.des.knora_u import KNORAU
from deslib.des.meta_des import METADES
# Example of stacked model
from deslib.static.stacked import StackedClassifier

rng = np.random.RandomState(42)

# Fetch a classification dataset from OpenML
data = fetch_openml(name='credit-g', cache=False, as_frame=False)
X = data.data
y = data.target
# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
                                                    random_state=rng)
RF = RandomForestClassifier(random_state=rng)
RF.fit(X_train, y_train)

X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train,
                                                    test_size=0.50,
                                                    random_state=rng)

# Training a random forest to be used as the pool of classifiers.
# We set the maximum depth of the tree so that it
# can estimate probabilities
pool_classifiers = RandomForestClassifier(n_estimators=100, max_depth=5,
```

(continues on next page)

(continued from previous page)

```
                                            random_state=rng)
pool_classifiers.fit(X_train, y_train)

stacked = StackedClassifier(pool_classifiers, LogisticRegression())
stacked.fit(X_dsel, y_dsel)

# Initialize a DS technique. Here we specify the size of
# the region of competence (5 neighbors)
knorau = KNORAU(pool_classifiers, random_state=rng)
kne = KNORAE(pool_classifiers, k=5, random_state=rng)
desp = DESP(pool_classifiers, k=5, random_state=rng)
ola = OLA(pool_classifiers, k=5, random_state=rng)
mcb = MCB(pool_classifiers, k=5, random_state=rng)
meta = METADES(pool_classifiers, k=5, random_state=rng)

# Fit the DS techniques
knorau.fit(X_dsel, y_dsel)
kne.fit(X_dsel, y_dsel)
desp.fit(X_dsel, y_dsel)
meta.fit(X_dsel, y_dsel)
ola.fit(X_dsel, y_dsel)
mcb.fit(X_dsel, y_dsel)
```

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/deslib/envs/v0.3.5/lib/python3.6/
↪site-packages/sklearn/datasets/_openml.py:407: UserWarning: Multiple active␣
↪versions of the dataset matching the name credit-g exist. Versions may be␣
↪fundamentally different, returning version 1.
  " {version}.".format(name=name, version=res[0]['version']))

MCB(k=5,
    pool_classifiers=RandomForestClassifier(max_depth=5,
                                            random_state=RandomState(MT19937) at␣
↪0x7F5C0FDAA888),
    random_state=RandomState(MT19937) at 0x7F5C0FDAA888)
```

Plotting the results

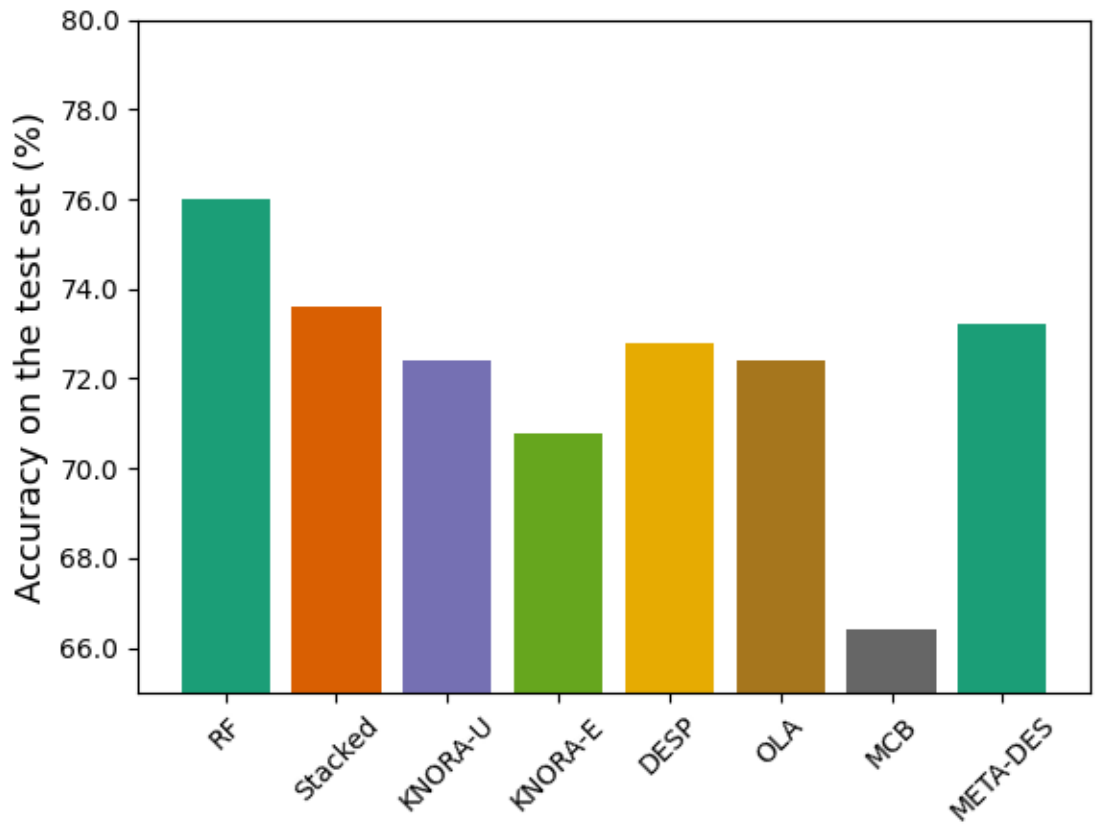
Let's now evaluate the methods on the test set.

```
rf_score = RF.score(X_test, y_test)
stacked_score = stacked.score(X_test, y_test)
knorau_score = knorau.score(X_test, y_test)
kne_score = kne.score(X_test, y_test)
desp_score = desp.score(X_test, y_test)
ola_score = ola.score(X_test, y_test)
mcb_score = mcb.score(X_test, y_test)
meta_score = meta.score(X_test, y_test)
print('Classification accuracy RF: ', rf_score)
print('Classification accuracy Stacked: ', stacked_score)
print('Evaluating DS techniques:')
print('Classification accuracy KNORA-U: ', knorau_score)
print('Classification accuracy KNORA-E: ', kne_score)
print('Classification accuracy DESP: ', desp_score)
```

(continues on next page)

(continued from previous page)

```
print('Classification accuracy OLA: ', ola_score)
print('Classification accuracy MCB: ', mcb_score)
print('Classification accuracy META-DES: ', meta_score)

cmap = get_cmap('Dark2')
colors = [cmap(i) for i in np.linspace(0, 1, 7)]
labels = ['RF', 'Stacked', 'KNORA-U', 'KNORA-E', 'DESP', 'OLA', 'MCB',
          'META-DES']

fig, ax = plt.subplots()
pct_formatter = FuncFormatter(lambda x, pos: '{:.1f}'.format(x * 100))
ax.bar(np.arange(8),
       [rf_score, stacked_score, knorau_score, kne_score, desp_score,
        ola_score, mcb_score, meta_score],
       color=colors,
       tick_label=labels)
ax.set_ylim(0.65, 0.80)
ax.set_xlabel('Method', fontsize=13)
ax.set_ylabel('Accuracy on the test set (%)', fontsize=13)
ax.yaxis.set_major_formatter(pct_formatter)
for tick in ax.get_xticklabels():
    tick.set_rotation(45)
plt.subplots_adjust(bottom=0.15)
plt.show()
```



Out:

```
Classification accuracy RF:  0.76
Classification accuracy Stacked:  0.736
Evaluating DS techniques:
Classification accuracy KNORA-U:  0.724
Classification accuracy KNORA-E:  0.708
Classification accuracy DESP:  0.728
Classification accuracy OLA:  0.724
Classification accuracy MCB:  0.664
Classification accuracy META-DES:  0.732
```

Total running time of the script: (0 minutes 8.467 seconds)

3.3.8 Example using heterogeneous ensemble

DESlib accepts different classifier models in the pool of classifiers. Such pool of classifiers is called Heterogeneous.

In this example, we consider a pool of classifiers composed of a Gaussian Naive Bayes, Perceptron, k-NN, Decision tree and Gaussian SVM. We also compare the result of DS methods with the voting classifier from sklearn.

```
import numpy as np
from sklearn.calibration import CalibratedClassifierCV
# Importing dataset and preprocessing routines
from sklearn.datasets import fetch_openml
from sklearn.ensemble import VotingClassifier
# Base classifier models:
from sklearn.linear_model import Perceptron
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

from deslib.dcs import MCB
# Example of DCS techniques
from deslib.dcs import OLA
from deslib.des import DESP
# Example of DES techniques
from deslib.des import KNORAE
from deslib.des import KNORAU
from deslib.des import METADES
from deslib.static import StackedClassifier

rng = np.random.RandomState(42)
data = fetch_openml(name='australian', cache=False, as_frame=False)
X = data.data
y = data.target

# split the data into training and test data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33,
                                                    random_state=rng)

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

(continues on next page)

(continued from previous page)

```
# Split the data into training and DSEL for DS techniques
X_train, X_dsel, y_train, y_dsel = train_test_split(X_train, y_train,
                                                    test_size=0.5,
                                                    random_state=rng)

model_perceptron = CalibratedClassifierCV(Perceptron(max_iter=100,
                                                     random_state=rng),
                                          cv=3)

model_perceptron.fit(X_train, y_train)
model_svc = SVC(probability=True, gamma='auto',
                random_state=rng).fit(X_train, y_train)
model_bayes = GaussianNB().fit(X_train, y_train)
model_tree = DecisionTreeClassifier(random_state=rng).fit(X_train, y_train)
model_knn = KNeighborsClassifier(n_neighbors=1).fit(X_train, y_train)

pool_classifiers = [model_perceptron,
                    model_svc,
                    model_bayes,
                    model_tree,
                    model_knn]

voting_classifiers = [("perceptron", model_perceptron),
                      ("svc", model_svc),
                      ("bayes", model_bayes),
                      ("tree", model_tree),
                      ("knn", model_knn)]

model_voting = VotingClassifier(estimators=voting_classifiers).fit(
    X_train, y_train)

# Initializing the techniques
knorau = KNORAU(pool_classifiers)
kne = KNORAE(pool_classifiers)
desp = DESP(pool_classifiers)
metades = METADES(pool_classifiers, mode='hybrid')
# DCS techniques
ola = OLA(pool_classifiers)
mcb = MCB(pool_classifiers)
```

Adding stacked classifier as baseline comparison. Stacked classifier can be found in the static module. In this experiment we consider two types of stacking: one using logistic regression as meta-classifier (default configuration) and the other using a Decision Tree.

```
stacked_lr = StackedClassifier(pool_classifiers, random_state=rng)
stacked_dt = StackedClassifier(pool_classifiers,
                               random_state=rng,
                               meta_classifier=DecisionTreeClassifier())
# Fitting the DS techniques
knorau.fit(X_dsel, y_dsel)
kne.fit(X_dsel, y_dsel)
desp.fit(X_dsel, y_dsel)
metades.fit(X_dsel, y_dsel)
ola.fit(X_dsel, y_dsel)
mcb.fit(X_dsel, y_dsel)
```

(continues on next page)

(continued from previous page)

```
# Fitting the tacking models
stacked_lr.fit(X_dsel, y_dsel)
stacked_dt.fit(X_dsel, y_dsel)

# Calculate classification accuracy of each technique
print('Evaluating DS techniques:')
print('Classification accuracy of Majority voting the pool: ',
      model_voting.score(X_test, y_test))
print('Classification accuracy of KNORA-U: ', knorau.score(X_test, y_test))
print('Classification accuracy of KNORA-E: ', kne.score(X_test, y_test))
print('Classification accuracy of DESP: ', desp.score(X_test, y_test))
print('Classification accuracy of META-DES: ', metades.score(X_test, y_test))
print('Classification accuracy of OLA: ', ola.score(X_test, y_test))
print('Classification accuracy Stacking LR', stacked_lr.score(X_test, y_test))
print('Classification accuracy Stacking DT', stacked_dt.score(X_test, y_test))
```

Total running time of the script: (0 minutes 0.000 seconds)

3.3.9 Dynamic selection with linear classifiers: XOR example

This example shows that DS can deal with non-linear problem (XOR) using a combination of a few linear base classifiers.

- 10 dynamic selection methods (5 DES and 5 DCS) are evaluated with a pool composed of Decision stumps.
- Since we use Bagging to generate the base classifiers, we also included its performance as a baseline comparison.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.ensemble import BaggingClassifier
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

from deslib.dcs import LCA
from deslib.dcs import MLA
from deslib.dcs import OLA
from deslib.dcs import MCB
from deslib.dcs import Rank

from deslib.des import DESKNN
from deslib.des import KNORAE
from deslib.des import KNORAU
from deslib.des import KNOP
from deslib.des import METADES
from deslib.util.datasets import make_xor
```

Defining helper functions to facilitate plotting the decision boundaries:

```
def plot_classifier_decision(ax, clf, X, mode='line', **params):

    xx, yy = make_grid(X[:, 0], X[:, 1])

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    if mode == 'line':
```

(continues on next page)

(continued from previous page)

```
        ax.contour(xx, yy, Z, **params)
    else:
        ax.contourf(xx, yy, Z, **params)
    ax.set_xlim((np.min(X[:, 0]), np.max(X[:, 0])))
    ax.set_ylim((np.min(X[:, 1]), np.max(X[:, 0])))


def plot_dataset(X, y, ax=None, title=None, **params):

    if ax is None:
        ax = plt.gca()
    ax.scatter(X[:, 0], X[:, 1], marker='o', c=y, s=25,
               edgecolor='k', **params)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
    if title is not None:
        ax.set_title(title)
    return ax


def make_grid(x, y, h=.02):

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


# Prepare the DS techniques. Changing k value to 5.
def initialize_ds(pool_classifiers, X, y, k=5):
    knorau = KNORAU(pool_classifiers, k=k)
    kne = KNORAE(pool_classifiers, k=k)
    desknn = DESKNN(pool_classifiers, k=k)
    ola = OLA(pool_classifiers, k=k)
    lca = LCA(pool_classifiers, k=k)
    mla = MLA(pool_classifiers, k=k)
    mcb = MCB(pool_classifiers, k=k)
    rank = Rank(pool_classifiers, k=k)
    knop = KNOP(pool_classifiers, k=k)
    meta = METADES(pool_classifiers, k=k)

    list_ds = [knorau, kne, ola, lca, mla, desknn, mcb, rank, knop, meta]
    names = ['KNORA-U', 'KNORA-E', 'OLA', 'LCA', 'MLA', 'DESKNN', 'MCB',
             'RANK', 'KNOP', 'META-DES']
    # fit the ds techniques
    for ds in list_ds:
        ds.fit(X, y)

    return list_ds, names
```

Generating the dataset and training the pool of classifiers.

```
rng = np.random.RandomState(1234)
X, y = make_xor(1000, random_state=rng)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
                                                    random_state=rng)
```

(continues on next page)

(continued from previous page)

```
X_DSEL, X_test, y_DSEL, y_test = train_test_split(X_train, y_train,
                                                  test_size=0.5,
                                                  random_state=rng)

pool_classifiers = BaggingClassifier(DecisionTreeClassifier(max_depth=1),
                                     n_estimators=10,
                                     random_state=rng)
pool_classifiers.fit(X_train, y_train)
```

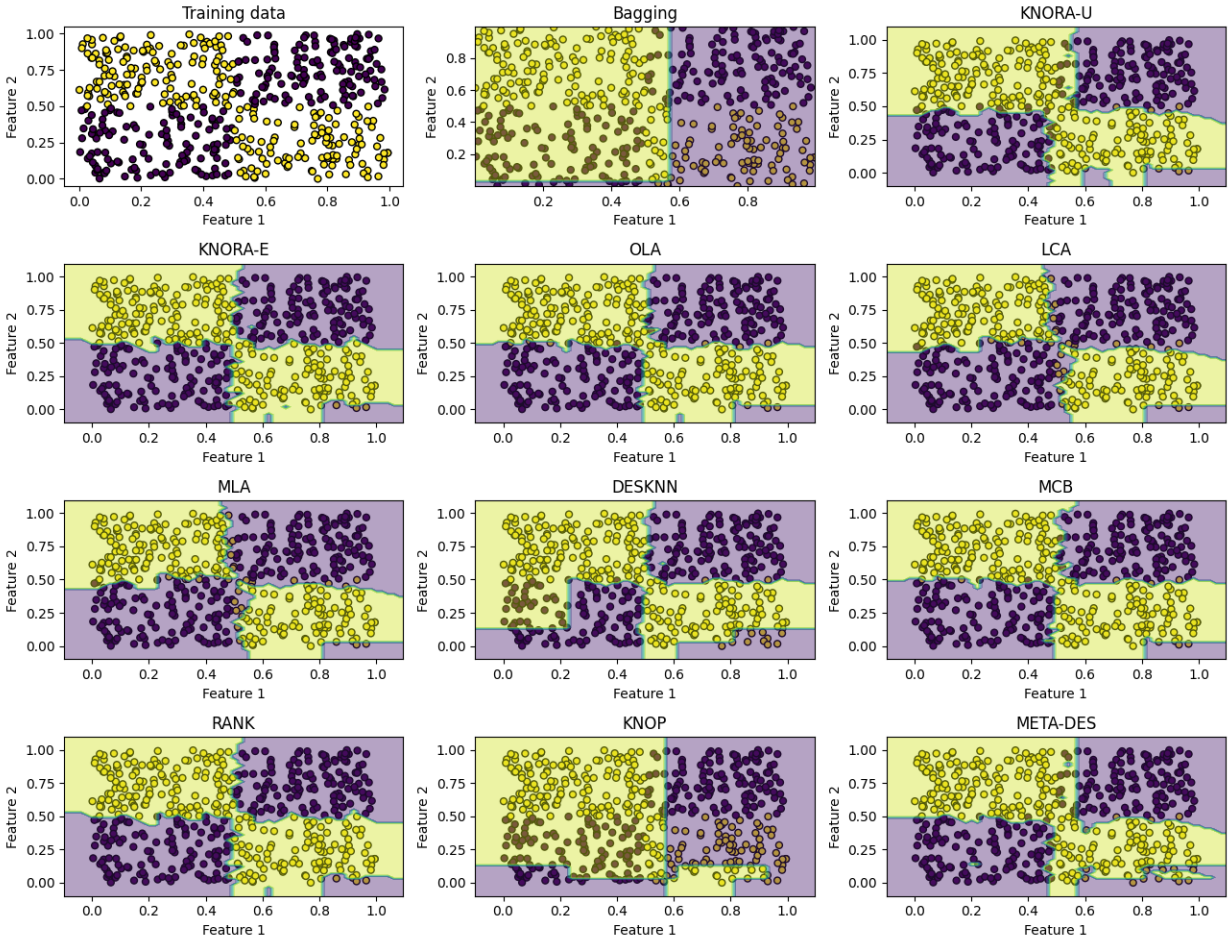
Out:

```
BaggingClassifier(base_estimator=DecisionTreeClassifier(max_depth=1),
                  random_state=RandomState(MT19937) at 0x7F5C0FDAA780)
```

Merging training and validation data to compose DSEL

In this example merge the training data with the validation, to create a DSEL having more examples for the competence estimation. Using the training data for dynamic selection can be beneficial when dealing with small sample size datasets. However, in this case we need to have a pool composed of weak classifier so that the base classifiers are not able to memorize the training data (overfit).

```
X_DSEL = np.vstack((X_DSEL, X_train))
y_DSEL = np.hstack((y_DSEL, y_train))
list_ds, names = initialize_ds(pool_classifiers, X_DSEL, y_DSEL, k=7)

fig, sub = plt.subplots(4, 3, figsize=(13, 10))
plt.subplots_adjust(wspace=0.4, hspace=0.4)

ax_data = sub.flatten()[0]
ax_bagging = sub.flatten()[1]
plot_dataset(X_train, y_train, ax=ax_data, title="Training data")

plot_dataset(X_train, y_train, ax=ax_bagging)
plot_classifier_decision(ax_bagging, pool_classifiers,
                         X_train, mode='filled', alpha=0.4)
ax_bagging.set_title("Bagging")

# Plotting the decision border of the DS methods
for ds, name, ax in zip(list_ds, names, sub.flatten()[2:]):
    plot_dataset(X_train, y_train, ax=ax)
    plot_classifier_decision(ax, ds, X_train, mode='filled', alpha=0.4)
    ax.set_xlim((np.min(X_train[:, 0]) - 0.1, np.max(X_train[:, 0] + 0.1)))
    ax.set_ylim((np.min(X_train[:, 1]) - 0.1, np.max(X_train[:, 1] + 0.1)))
    ax.set_title(name)
plt.show()
plt.tight_layout()
```



Evaluation on the test set

Finally, let's evaluate the classification accuracy of DS techniques and Bagging on the test set:

```
for ds, name in zip(list_ds, names):
    print('Accuracy ' + name + ': ' + str(ds.score(X_test, y_test)))
print('Accuracy Bagging: ' + str(pool_classifiers.score(X_test, y_test)))
```

Out:

```
Accuracy KNORA-U: 0.92
Accuracy KNORA-E: 0.992
Accuracy OLA: 0.976
Accuracy LCA: 0.944
Accuracy MLA: 0.944
Accuracy DESKNN: 0.908
Accuracy MCB: 0.968
Accuracy RANK: 0.992
Accuracy KNOP: 0.636
Accuracy META-DES: 0.928
Accuracy Bagging: 0.584
```

Total running time of the script: (0 minutes 21.018 seconds)

3.3.10 Visualizing decision boundaries on the P2 problem

This example shows the power of dynamic selection (DS) techniques which can solve complex non-linear classification near classifiers. It also compares the performance of DS techniques with some baseline classification methods such as Random Forests, AdaBoost and SVMs.

The P2 is a two-class problem, presented by Valentini, in which each class is defined in multiple decision regions delimited by polynomial and trigonometric functions:

$$to$$

$$E1(x) = sin(x) + 5$$
$$E2(x) = (x-2)^2 + 1$$
$$E3(x) = -0.1 \cdot x^2 + 0.6sin(4x) + 8$$
$$E4(x) = \frac{(x-10)^2}{2} + 7.902(3.2)$$

It is impossible to solve this problem using a single linear classifier. The performance of the best possible linear classifier is around 50%.

Let's start by importing all required modules, and defining helper functions to facilitate plotting the decision boundaries:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.ensemble import AdaBoostClassifier, RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

# Importing DS techniques
from deslib.dcs.ola import OLA
from deslib.dcs.rank import Rank
from deslib.des.des_p import DESP
from deslib.des.knora_e import KNORAE
from deslib.static import StackedClassifier
from deslib.util.datasets import make_P2


# Plotting-related functions
def make_grid(x, y, h=.02):
    x_min, x_max = x.min() - 1, x.max() + 1
    y_min, y_max = y.min() - 1, y.max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))
```

(continues on next page)

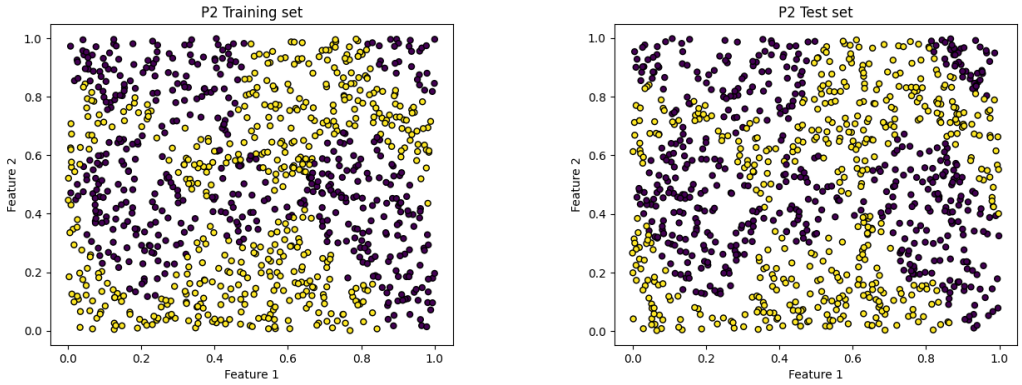
(continued from previous page)

```
    return xx, yy


def plot_classifier_decision(ax, clf, X, mode='line', **params):
    xx, yy = make_grid(X[:, 0], X[:, 1])

    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    if mode == 'line':
        ax.contour(xx, yy, Z, **params)
    else:
        ax.contourf(xx, yy, Z, **params)
    ax.set_xlim((np.min(X[:, 0]), np.max(X[:, 0])))
    ax.set_ylim((np.min(X[:, 1]), np.max(X[:, 0])))


def plot_dataset(X, y, ax=None, title=None, **params):
    if ax is None:
        ax = plt.gca()
    ax.scatter(X[:, 0], X[:, 1], marker='o', c=y, s=25,
               edgecolor='k', **params)
    ax.set_xlabel('Feature 1')
    ax.set_ylabel('Feature 2')
    if title is not None:
        ax.set_title(title)
    return ax
```

Visualizing the dataset

Now let's generate and plot the dataset:

```
# Generating and plotting the P2 Dataset:
rng = np.random.RandomState(1234)
X, y = make_P2([1000, 1000], random_state=rng)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5,
                                                    random_state=rng)
fig, axs = plt.subplots(1, 2, figsize=(15, 5))
plt.subplots_adjust(wspace=0.4, hspace=0.4)
plot_dataset(X_train, y_train, ax=axs[0], title='P2 Training set')
plot_dataset(X_test, y_test, ax=axs[1], title='P2 Test set')
```



Out:

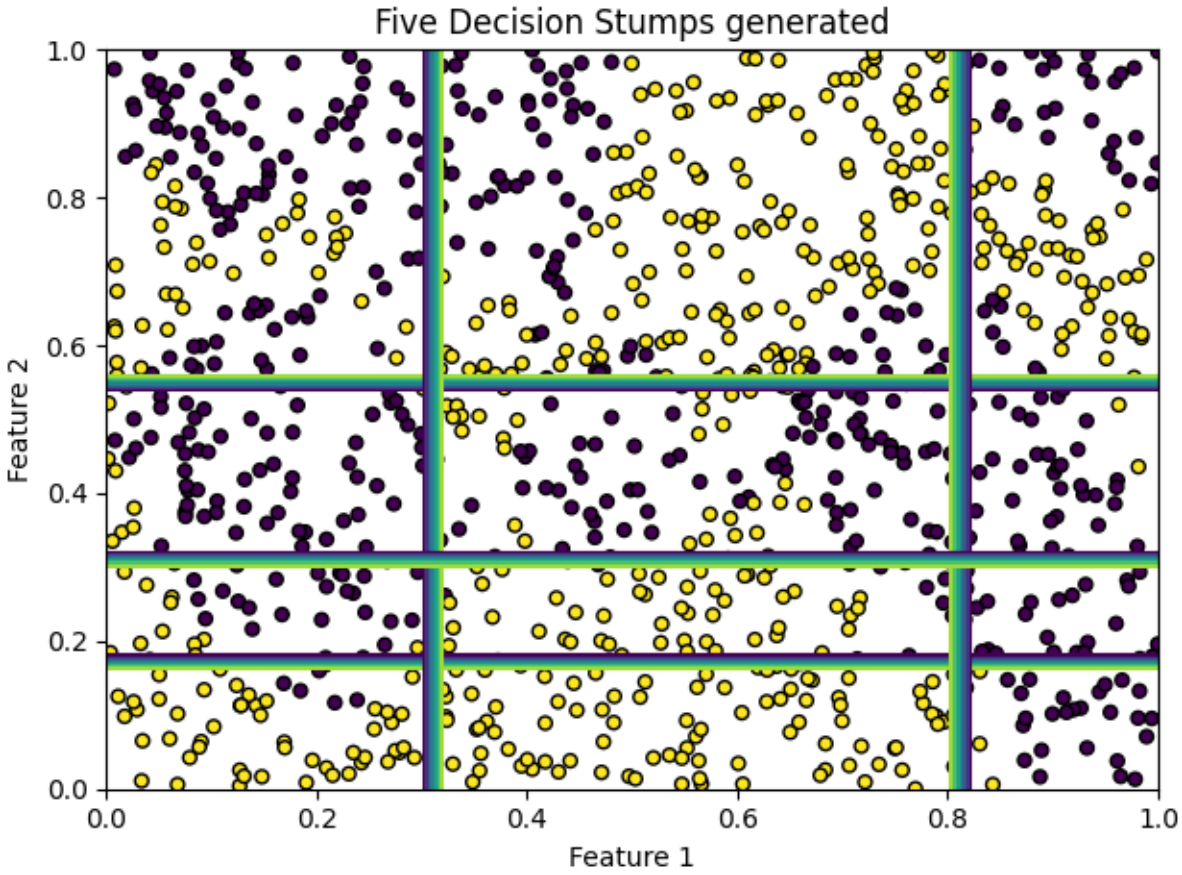
```
<AxesSubplot:title={'center':'P2 Test set'}, xlabel='Feature 1', ylabel='Feature 2'>
```

Evaluating the performance of dynamic selection methods

We will now generate a pool composed of 5 Decision Stumps using AdaBoost.

These are weak linear models. Each base classifier has a classification performance close to 50%.

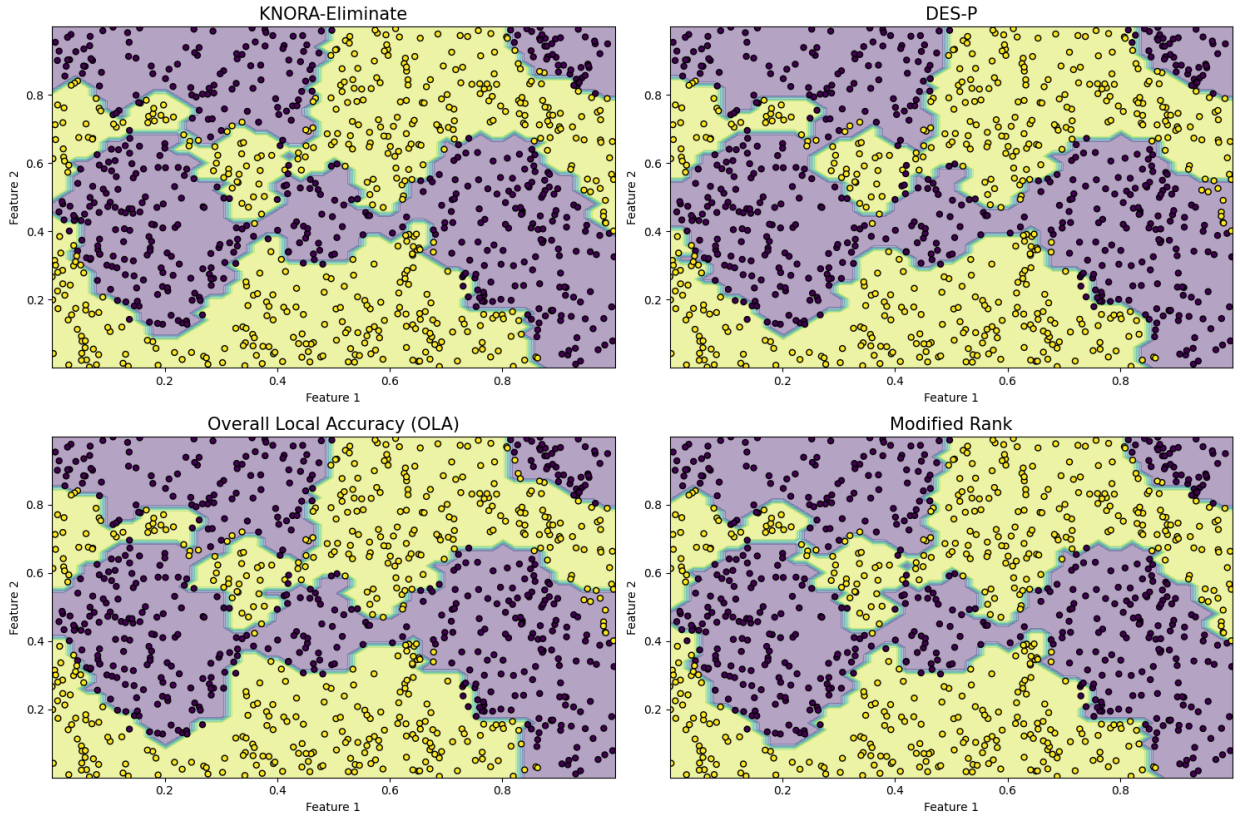
```
pool_classifiers = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1),
                                      n_estimators=5, random_state=rng)
pool_classifiers.fit(X_train, y_train)

ax = plot_dataset(X_train, y_train, title='Five Decision Stumps generated')
for clf in pool_classifiers:
    plot_classifier_decision(ax, clf, X_train)
    ax.set_xlim((0, 1))
    ax.set_ylim((0, 1))

plt.show()
plt.tight_layout()
```



Comparison with Dynamic Selection techniques

We will now consider four DS methods: k-Nearest Oracle-Eliminate (KNORA-E), Dynamic Ensemble Selection performance (DES-P), Overall Local Accuracy (OLA) and Rank. Let's train the classifiers and plot their decision boundaries:

```
knora_e = KNORAE(pool_classifiers).fit(X_train, y_train)
desp = DESP(pool_classifiers).fit(X_train, y_train)
ola = OLA(pool_classifiers).fit(X_train, y_train)
rank = Rank(pool_classifiers).fit(X_train, y_train)

# Plotting the Decision Border of the DS methods.
fig2, sub = plt.subplots(2, 2, figsize=(15, 10))
plt.subplots_adjust(wspace=0.4, hspace=0.4)
titles = ['KNORA-Eliminate', 'DES-P', 'Overall Local Accuracy (OLA)',
          'Modified Rank']

classifiers = [knora_e, desp, ola, rank]
for clf, ax, title in zip(classifiers, sub.flatten(), titles):
    plot_classifier_decision(ax, clf, X_train, mode='filled', alpha=0.4)
    plot_dataset(X_test, y_test, ax=ax)
    ax.set_xlim(np.min(X[:, 0]), np.max(X[:, 0]))
    ax.set_ylim(np.min(X[:, 1]), np.max(X[:, 1]))
    ax.set_title(title, fontsize=15)

# Setting figure to show
# sphinx_gallery_thumbnail_number = 3

plt.show()
plt.tight_layout()
```



Comparison to baselines

Let's now compare the results with four baselines: Support Vector Machine (SVM) with an RBF kernel; Multi-Layer Perceptron (MLP), Random Forest, Adaboost, and Stacking.

```
# Setting a baseline using standard classification methods
svm = SVC(gamma='scale', random_state=rng).fit(X_train, y_train)
mlp = MLPClassifier(max_iter=10000, random_state=rng).fit(X_train, y_train)
forest = RandomForestClassifier(n_estimators=10,
                                random_state=rng).fit(X_train, y_train)
boosting = AdaBoostClassifier(random_state=rng).fit(X_train, y_train)
stacked_lr = StackedClassifier(pool_classifiers=pool_classifiers,
                               random_state=rng)
stacked_lr.fit(X_train, y_train)

stacked_dt = StackedClassifier(pool_classifiers=pool_classifiers,
                               random_state=rng,
                               meta_classifier=DecisionTreeClassifier())
stacked_dt.fit(X_train, y_train)
```

Out:

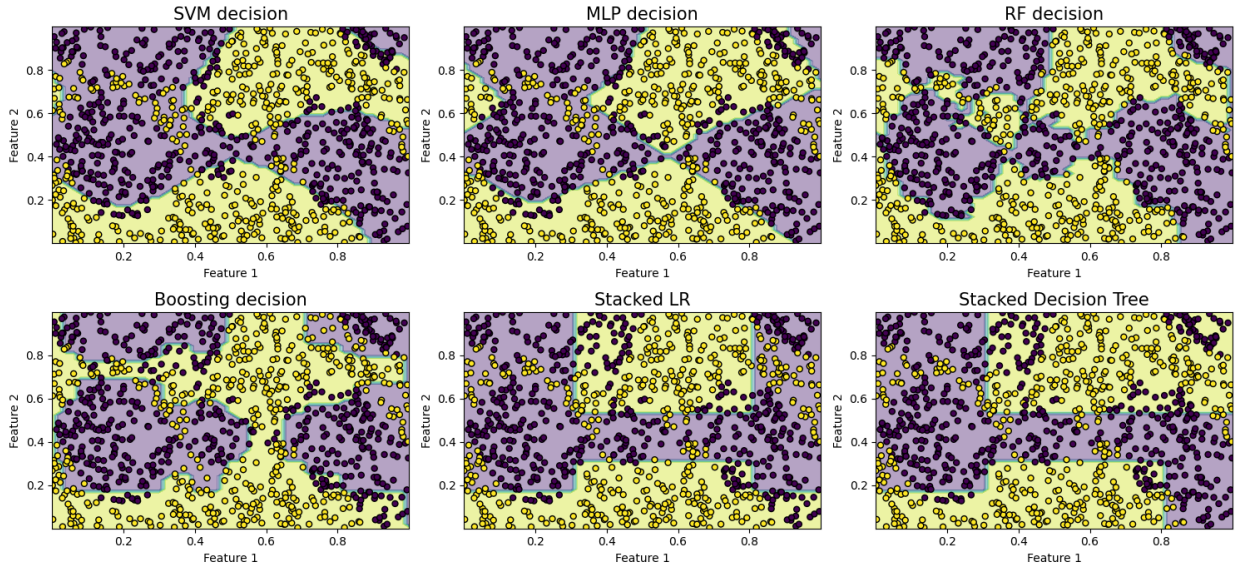
```
StackedClassifier(meta_classifier=DecisionTreeClassifier(),
                  pool_classifiers=AdaBoostClassifier(base_
↪estimator=DecisionTreeClassifier(max_depth=1),
                                                      n_estimators=5,
                                                      random_
↪state=RandomState(MT19937) at 0x7F5C0FDAACA8),
```

(continues on next page)

(continued from previous page)

```
                    random_state=RandomState(MT19937) at 0x7F5C0FDAACA8)
```

```
fig2, sub = plt.subplots(2, 3, figsize=(15, 7))
plt.subplots_adjust(wspace=0.4, hspace=0.4)
titles = ['SVM decision', 'MLP decision', 'RF decision',
          'Boosting decision', 'Stacked LR', 'Stacked Decision Tree']
classifiers = [svm, mlp, forest, boosting, stacked_lr, stacked_dt]
for clf, ax, title in zip(classifiers, sub.flatten(), titles):
    plot_classifier_decision(ax, clf, X_test, mode='filled', alpha=0.4)
    plot_dataset(X_test, y_test, ax=ax)
    ax.set_xlim(np.min(X[:, 0]), np.max(X[:, 0]))
    ax.set_ylim(np.min(X[:, 1]), np.max(X[:, 1]))
    ax.set_title(title, fontsize=15)

plt.show()
plt.tight_layout()
```



Evaluation on the test set

Finally, let's evaluate the baselines and the Dynamic Selection methods on the test set:

```
print('KNORAE score = {}'.format(knora_e.score(X_test, y_test)))
print('DESP score = {}'.format(desp.score(X_test, y_test)))
print('OLA score = {}'.format(ola.score(X_test, y_test)))
print('Rank score = {}'.format(rank.score(X_test, y_test)))
print('SVM score = {}'.format(svm.score(X_test, y_test)))
print('MLP score = {}'.format(mlp.score(X_test, y_test)))
print('RF score = {}'.format(forest.score(X_test, y_test)))
print('Boosting score = {}'.format(boosting.score(X_test, y_test)))
print('Stacking LR score = {}' .format(stacked_lr.score(X_test, y_test)))
print('Staking Decision Tree = {}' .format(stacked_dt.score(X_test, y_test)))
```

Out:

```
KNORAE score = 0.948
DESP score = 0.927
OLA score = 0.932
Rank score = 0.948
SVM score = 0.798
MLP score = 0.794
RF score = 0.923
Boosting score = 0.795
Stacking LR score = 0.716
Staking Decision Tree = 0.732
```

Total running time of the script: (0 minutes 19.248 seconds)

3.4 Release history

3.4.1 Version 0.3

- Third release of the stable API. By Rafael M O Cruz and Luiz G Hafemann

Changes

- All techniques are now sklearn estimators and passes the check_estimator tests.
- All techniques can now be instantiated without a trained pool of classifiers.
- Pool of classifiers can now be fitted together with the ensemble techniques. See simple example.
- Added support for Faiss (Facebook AI Similarity Search) for fast region of competence estimation on GPU.
- Added DES Multi-class Imbalance method *deslib.des.des_mi.DESMI*.
- Added stacked classifier model, *deslib.static.stacked.StackedClassifier* to the static ensemble module.
- Added a new Instance Hardness measure `utils.instance_hardness.kdn_score()`.
- Added Instance Hardness support when using DES-Clustering.
- Added label encoder for the `static` module.
- Added a script `utils.datasets` with routines to generate synthetic datasets (e.g., the P2 and XOR datasets).
- Changed name of base classes (Adding Base to their following scikit-learn standards).
- Removal of **DFP_mask**, **neighbors** and **distances** as class variables.
- Changed signature of methods **estimate_competence**, **predict_with_ds**, **predict_proba_with_ds**. They now require the neighbors and distances to be passed as input arguments.
- Added random_state parameter to all methods in order to have reproducible results.
- Added Python 3.7 support.
- New and updated examples.
- Added performance tests comparing the speed of Faiss vs sklearn KNN.

Bug Fixes

- Fixed bug with META-DES when checking if the meta-classifier was already fitted.
- Fixed bug with random state on DCS techniques.
- Fixed high memory consumption on DES probabilistic methods.
- Fixed bug on Heterogeneous ensembles example and notebooks examples.
- Fixed bug on *deslib.des.probabilistic.MinimumDifference* when only samples from a single class are provided.
- Fixed problem with DS methods when the number of training examples was lower than the k value.
- Fixed division by zero problems with `APosteriori` `APriori` `MLA` when the distance is equal to zero.
- Fixed bug on `deslib.utils.prob_functions.exponential_func()` when the support obtained for the correct class was equal to one.

3.4.2 Version 0.2

- Second release of the stable API. By Rafael M O Cruz and Luiz G Hafemann.

Changes

- Implemented Label Encoding: labels are no longer required to be integers starting from 0. Categorical (strings) and non-sequential integers are supported (similarly to scikit-learn).
- Batch processing: Vectorized implementation of predictions. Large speed-up in computation time (100x faster in some cases).
- Predict proba: only required (in the base estimators) if using methods that rely on probabilities (or if requesting probabilities from the ensemble).
- Improved documentation: Included additional examples, a step-by-step tutorial on how to use the library.
- New integration tests: Now covering predict_proba, IH and DFP.
- Bug fixes on 1) predict_proba 2) KNOP with DFP.

3.4.3 Version 0.1

API

- First release of the stable API. By Rafael M O Cruz and Luiz G Hafemann.

Implemented methods:

- **DES techniques currently available are:**
    1. META-DES
    2. K-Nearest-Oracle-Eliminate (KNORA-E)
    3. K-Nearest-Oracle-Union (KNORA-U)
    4. Dynamic Ensemble Selection-Performance(DES-P)

5. K-Nearest-Output Profiles (KNOP)
6. Randomized Reference Classifier (DES-RRC)
7. DES Kullback-Leibler Divergence (DES-KL)
8. DES-Exponential
9. DES-Logarithmic
10. DES-Minimum Difference
11. DES-Clustering
12. DES-KNN

- **DCS techniques:**
    1. Modified Classifier Rank (Rank)
    2. Overall Locall Accuracy (OLA)
    3. Local Class Accuracy (LCA)
    4. Modified Local Accuracy (MLA)
    5. Multiple Classifier Behaviour (MCB)
    6. A Priori Selection (A Priori)
    7. A Posteriori Selection (A Posteriori)
- **Baseline methods:**
    1. Oracle
    2. Single Best
    3. Static Selection
- Dynamic Frienemy Prunning (DFP)
- Diversity measures
- Aggregation functions

Version 0.1

API

- First release of the stable API. By Rafael M O Cruz and Luiz G Hafemann.

Implemented methods:

- **DES techniques currently available are:**
    1. META-DES
    2. K-Nearest-Oracle-Eliminate (KNORA-E)
    3. K-Nearest-Oracle-Union (KNORA-U)
    4. Dynamic Ensemble Selection-Performance(DES-P)
    5. K-Nearest-Output Profiles (KNOP)

6. Randomized Reference Classifier (DES-RRC)
7. DES Kullback-Leibler Divergence (DES-KL)
8. DES-Exponential
9. DES-Logarithmic
10. DES-Minimum Difference
11. DES-Clustering
12. DES-KNN

- **DCS techniques:**
  1. Modified Classifier Rank (Rank)
  2. Overall Locall Accuracy (OLA)
  3. Local Class Accuracy (LCA)
  4. Modified Local Accuracy (MLA)
  5. Multiple Classifier Behaviour (MCB)
  6. A Priori Selection (A Priori)
  7. A Posteriori Selection (A Posteriori)
- **Baseline methods:**
  1. Oracle
  2. Single Best
  3. Static Selection
- Dynamic Frienemy Prunning (DFP)
- Diversity measures
- Aggregation functions

Version 0.2

- Second release of the stable API. By Rafael M O Cruz and Luiz G Hafemann.

Changes

- Implemented Label Encoding: labels are no longer required to be integers starting from 0. Categorical (strings) and non-sequential integers are supported (similarly to scikit-learn).
- Batch processing: Vectorized implementation of predictions. Large speed-up in computation time (100x faster in some cases).
- Predict proba: only required (in the base estimators) if using methods that rely on probabilities (or if requesting probabilities from the ensemble).
- Improved documentation: Included additional examples, a step-by-step tutorial on how to use the library.
- New integration tests: Now covering predict_proba, IH and DFP.
- Bug fixes on 1) predict_proba 2) KNOP with DFP.

Version 0.3

- Third release of the stable API. By Rafael M O Cruz and Luiz G Hafemann

Changes

- All techniques are now sklearn estimators and passes the check_estimator tests.
- All techniques can now be instantiated without a trained pool of classifiers.
- Pool of classifiers can now be fitted together with the ensemble techniques. See simple example.
- Added support for Faiss (Facebook AI Similarity Search) for fast region of competence estimation on GPU.
- Added DES Multi-class Imbalance method *deslib.des.des_mi.DESMI*.
- Added stacked classifier model, *deslib.static.stacked.StackedClassifier* to the static ensemble module.
- Added a new Instance Hardness measure `utils.instance_hardness.kdn_score()`.
- Added Instance Hardness support when using DES-Clustering.
- Added label encoder for the `static` module.
- Added a script `utils.datasets` with routines to generate synthetic datasets (e.g., the P2 and XOR datasets).
- Changed name of base classes (Adding Base to their following scikit-learn standards).
- Removal of **DFP_mask**, **neighbors** and **distances** as class variables.
- Changed signature of methods **estimate_competence**, **predict_with_ds**, **predict_proba_with_ds**. They now require the neighbors and distances to be passed as input arguments.
- Added random_state parameter to all methods in order to have reproducible results.
- Added Python 3.7 support.
- New and updated examples.
- Added performance tests comparing the speed of Faiss vs sklearn KNN.

Bug Fixes

- Fixed bug with META-DES when checking if the meta-classifier was already fitted.
- Fixed bug with random state on DCS techniques.
- Fixed high memory consumption on DES probabilistic methods.
- Fixed bug on Heterogeneous ensembles example and notebooks examples.
- Fixed bug on *deslib.des.probabilistic.MinimumDifference* when only samples from a single class are provided.
- Fixed problem with DS methods when the number of training examples was lower than the k value.
- Fixed division by zero problems with `APosteriori` `APriori` `MLA` when the distance is equal to zero.
- Fixed bug on `deslib.utils.prob_functions.exponential_func()` when the support obtained for the correct class was equal to one.

CHAPTER 4

Example

Here we present an example of the KNORA-E techniques using a random forest to generate the pool of classifiers:

```
from sklearn.ensemble import RandomForestClassifier
from deslib.des.knora_e import KNORAE

# Train a pool of 10 classifiers
pool_classifiers = RandomForestClassifier(n_estimators=10)
pool_classifiers.fit(X_train, y_train)

# Initialize the DES model
knorae = KNORAE(pool_classifiers)

# Preprocess the Dynamic Selection dataset (DSEL)
knorae.fit(X_dsel, y_dsel)

# Predict new examples:
knorae.predict(X_test)
```

The library accepts any list of classifiers (from scikit-learn) as input, including a list containing different classifier models (heterogeneous ensembles). More examples to use the API can be found in the examples page.

CHAPTER 5

Citation

If you use DESLib in a scientific paper, please consider citing the following paper:

Rafael M. O. Cruz, Luiz G. Hafemann, Robert Sabourin and George D. C. Cavalcanti **DESlib: A Dynamic ensemble selection library in Python.** arXiv preprint arXiv:1802.04967 (2018).

```
@article{JMLR:v21:18-144,
    author  = {Rafael M. O. Cruz and Luiz G. Hafemann and Robert Sabourin and George␣
↪D. C. Cavalcanti},
    title   = {DESlib: A Dynamic ensemble selection library in Python},
    journal = {Journal of Machine Learning Research},
    year    = {2020},
    volume  = {21},
    number  = {8},
    pages   = {1-5},
    url     = {http://jmlr.org/papers/v21/18-144.html}
}
```

5.1 References

Python Module Index

d

Index

F

H

K

L

M

N

O

P

Q

R

S

W