libact Documentation

Release 0.1.3

Y.-Y. Yang, S.-C. Lee, Y.-A. Chung, T.-E. Wu, S.-A. Chen, H.-T. Lin

Jun 04, 2018

Contents

libact is a python package designed to make active learning easier for real-world users. The package not only implements several popular active learning strategies, but also features the active learning by learning meta-strategy that allows the machine to automatically *learn* the best strategy on the fly. The package is designed for easy extension in terms of strategies, models and labelers. In particular, *libact* models can be easily obtained by interfacing with the models in scikit-learn.

CHAPTER 1

Table of Contents

1.1 Overview

libact is a Python package designed to make active learning easier for real-world users. The package not only implements several popular active learning strategies, but also features the active-learning-by-learning meta-algorithm that assists the users to automatically select the best strategy on the fly. Furthermore, the package provides a unified interface for implementing more strategies, models and application-specific labelers. The package is open-source along with issue trackers on github, and can be easily installed from Python Package Index repository.

Currently *libact* supports pool-based active learning problems, which consist of a set of labeled examples, a set of unlabeled examples, a supervised learning model, and a labeling oracle. In each iteration of active learning, the algorithm (also called a query strategy) queries the oracle to label an unlabeled example. The model can then be improved by the newly-labeled example. The goal is to use as few queries as possible for the model to achieve decent learning performance. Based on the components above, we have designed the following four interfaces for *libact*.

1.1.1 Dataset

A `libact.base.dataset.Dataset` object stores the labeled set and the unlabeled set. Each unlabeled or labeled example within a Dataset object is assigned with a unique identifier. After retrieving the label for an unlabeled example from the Labeler (the oracle to be discussed below), the update method is used to assign the label to the example, referenced by its identifier.

Internally, Dataset also maintains a callback queue. The on_update method can be used to register callback functions, which will be called after each update to the Dataset. The callback functions can be used for active learning algorithms that need to update their internal states after querying the oracle.

1.1.2 Labeler

A `libact.base.interfaces.Labeler` object plays the role of the oracle in the given active learning problem. Its label method takes in an unlabeled example and returns the retrieved label.

1.1.3 QueryStrategy

A *libact.base.interfaces.QueryStrategy* object implements an active learning algorithm. Each QueryStrategy object is associated with a Dataset object. When a QueryStrategy object is initialized, it will automatically register its update method as a callback function to the associated Dataset to be informed of any Dataset updates. The make_query method of a QueryStrategy object returns the identifier of an unlabeled example that the object (active learning algorithm) wants to query.

Currently, the following active learning algorithms are supported:

- Binary Classification
  - Density Weighted Uncertainty Sampling (density_weighted_uncertainty_sampling.py)
  - Hinted Sampling with SVM (hintsvm.py)
  - Query By Committee (query_by_committee.py)
  - Querying Informative and Representative Examples (quire.py)
  - Random Sampling (random_sampling.py)
  - Uncertainty Sampling (uncertainty_sampling.py)
  - Variance Reduction (variance_reduction.py)
- Multi-class Classification
  - Active Learning with Cost Embedding (multiclass/active_learning_with_cost_embedding.py)
  - Hierarchical Sampling (multiclass/hierarchical_sampling.py)
  - Expected Error Reduction (multiclass/expected_error_reduction.py)
  - Uncertainty Sampling (uncertainty_sampling.py)
- Multi-label Classification
  - Adaptive Active Learning (multilabel/adaptive_active_learning.py)
  - Binary Minimization (multilabel/binary_minimization.py)
  - Maximal Loss Reduction with Maximal Confidence (multilabel/maximum_margin_reduction.py)
  - Multi-label Active Learning With Auxiliary Learner (multilabel/multilabel_with_auxiliary_learner.py)

Note that because of legacy reasons, Uncertainty Sampling can handle multi-class setting though it is not under the multiclass submodule.

Additionally, we supported the *Active Learning By Learning* meta-algorithm (active_learning_by_learning.py) for selecting active learning algorithms for binary classification on the fly.

1.1.4 Model

A *libact.base.interfaces.Model* object represents a supervised classifiation algorithm. It contains train and predict methods, just like the fit and predict methods of the classification algorithms in scikit-learn. Note that the train method of Model only takes the labeled examples within Dataset for learning.

A *libact.base.interfaces.ContinuousModel* object represents an algorithm that supports continuous outputs during predictions, which includes an additional predict_real method.

Note that there is a `libact.models.SklearnAdapter` which takes a sklearn classifier instance and adaptes it to the libact Model interface.

1.1.5 Example Usage

Here is an example usage of *libact*:

```
# declare Dataset instance, X is the feature, y is the label (None if unlabeled)
dataset = Dataset(X, y)
query_strategy = QueryStrategy(dataset) # declare a QueryStrategy instance
labler = Labeler() # declare Labeler instance
model = Model() # declare model instance

for _ in range(quota): # loop through the number of queries
    query_id = query_strategy.make_query() # let the specified QueryStrategy suggest a data to query
    lbl = labeler.label(dataset.data[query_id][0]) # query the label of the example at query_id
    dataset.update(query_id, lbl) # update the dataset with newly-labeled example
    model.train(dataset) # train model with newly-updated Dataset
```

1.2 Examples

Here are some examples of using *libact*:

1.2.1 Comparing Different Query Strategies

Example file: `examples/plot.py`

This example shows the basic way to compare two active learning algorithm. The script is located in `/examples/plot.py`. Before running the script, you need to download sample dataset by running `/examples/get_dataset.py` and choose the one you want in variable `dataset_filepath`.

```
    # Specifiy the parameters here:
```

First, the data are splitted into training and testing set:

```
def split_train_test(dataset_filepath, test_size, n_labeled):
    X, y = import_libsvm_sparse(dataset_filepath).format_sklearn()

    X_train, X_test, y_train, y_test = \
        train_test_split(X, y, test_size=test_size)
    trn_ds = Dataset(X_train, np.concatenate(
        [y_train[:n_labeled], [None] * (len(y_train) - n_labeled)]))
    tst_ds = Dataset(X_test, y_test)
    fully_labeled_trn_ds = Dataset(X_train, y_train)

    return trn_ds, tst_ds, y_train, fully_labeled_trn_ds
```

The main part that uses *libact* is in the `run` function:

```
def run(trn_ds, tst_ds, lbr, model, qs, quota):
    E_in, E_out = [], []

    for _ in range(quota):
        # Standard usage of libact objects
        ask_id = qs.make_query()
```

(continues on next page)

(continued from previous page)

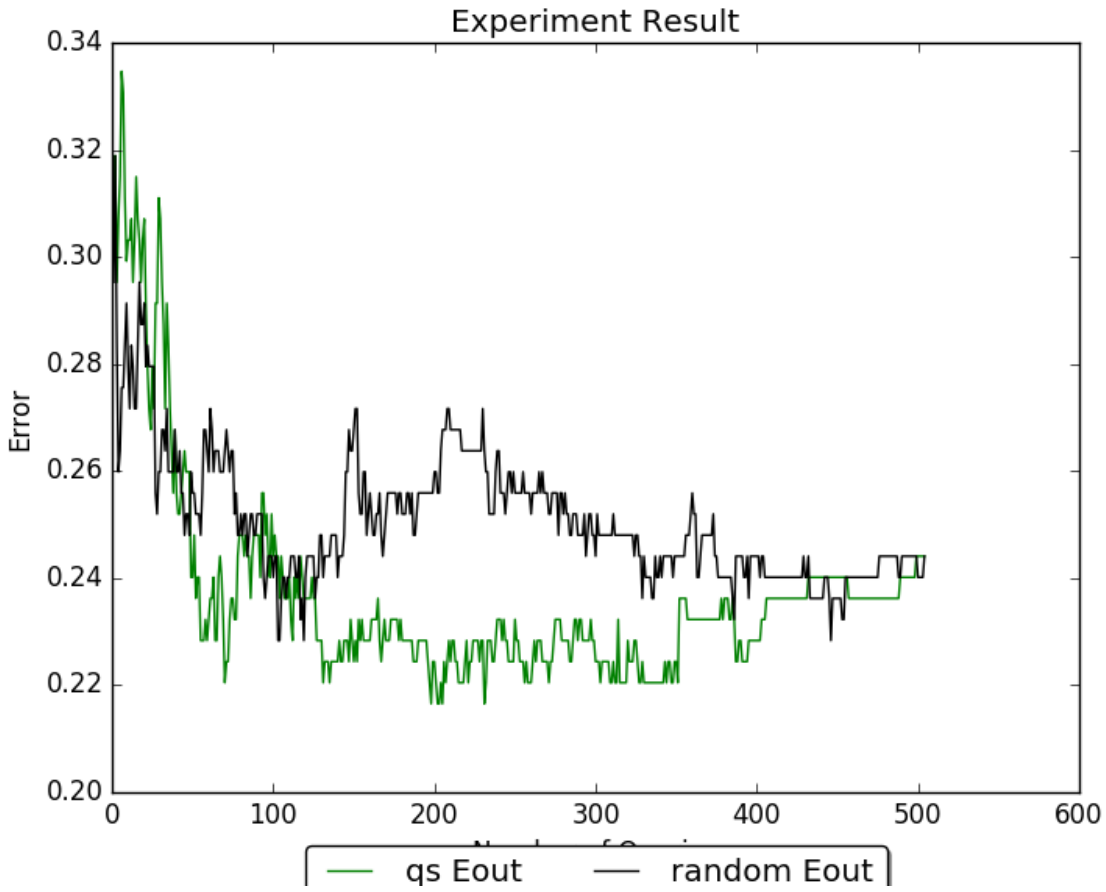
```
        X, _ = zip(*trn_ds.data)
        lb = lbr.label(X[ask_id])
        trn_ds.update(ask_id, lb)

        model.train(trn_ds)
        E_in = np.append(E_in, 1 - model.score(trn_ds))
        E_out = np.append(E_out, 1 - model.score(tst_ds))

    return E_in, E_out
```

In the `for` loop on line `25`, it iterates through each query in active learning process. `qs.make_query` returns the index of the sample that the active learning algorithm wants to query. `lbr` acts as the oracle and `lbr.label` returns the label of the given sample answered by oracle. `ds.update` updates the unlabeled sample with queried label.

A common way of evaluating the performance of active learning algorithm is to plot the learning curve. Where the X-axis is the number samples of queried, and the Y-axis is the corresponding error rate. List `E_in`, `E_out` collects the in-sample and out-sample error rate after each query. These information will be used to plot the learning curve. Learning curve are plotted by the following code:

```
    # Plot the learning curve of UncertaintySampling to RandomSampling
    # The x-axis is the number of queries, and the y-axis is the corresponding
    # error rate.
    query_num = np.arange(1, quota + 1)
    plt.plot(query_num, E_in_1, 'b', label='qs Ein')
    plt.plot(query_num, E_in_2, 'r', label='random Ein')
    plt.plot(query_num, E_out_1, 'g', label='qs Eout')
    plt.plot(query_num, E_out_2, 'k', label='random Eout')
    plt.xlabel('Number of Queries')
    plt.ylabel('Error')
    plt.title('Experiment Result')
```

The following figure are the result of using the *diabetes* dataset with `train_test_split` and `LogisticRegression`'s random_state set as 0, and `random.seed(0)`. The E_out line are removed for simplicity.



We can see from the example that uncertainty sample is able to reach lower error rate faster than random sampling.

Full source code:

```
#!/usr/bin/env python3
"""
The script helps guide the users to quickly understand how to use
libact by going through a simple active learning task with clear
descriptions.
"""

import copy
import os

import numpy as np
import matplotlib.pyplot as plt
try:
    from sklearn.model_selection import train_test_split
except ImportError:
    from sklearn.cross_validation import train_test_split

# libact classes
from libact.base.dataset import Dataset, import_libsvm_sparse
from libact.models import *
from libact.query_strategies import *
from libact.labelers import IdealLabeler
```

(continues on next page)

(continued from previous page)

```
23
24
25  def run(trn_ds, tst_ds, lbr, model, qs, quota):
26      E_in, E_out = [], []
27
28      for _ in range(quota):
29          # Standard usage of libact objects
30          ask_id = qs.make_query()
31          X, _ = zip(*trn_ds.data)
32          lb = lbr.label(X[ask_id])
33          trn_ds.update(ask_id, lb)
34
35          model.train(trn_ds)
36          E_in = np.append(E_in, 1 - model.score(trn_ds))
37          E_out = np.append(E_out, 1 - model.score(tst_ds))
38
39      return E_in, E_out
40
41
42  def split_train_test(dataset_filepath, test_size, n_labeled):
43      X, y = import_libsvm_sparse(dataset_filepath).format_sklearn()
44
45      X_train, X_test, y_train, y_test = \
46          train_test_split(X, y, test_size=test_size)
47      trn_ds = Dataset(X_train, np.concatenate(
48          [y_train[:n_labeled], [None] * (len(y_train) - n_labeled)]))
49      tst_ds = Dataset(X_test, y_test)
50      fully_labeled_trn_ds = Dataset(X_train, y_train)
51
52      return trn_ds, tst_ds, y_train, fully_labeled_trn_ds
53
54
55  def main():
56      # Specifiy the parameters here:
57      # path to your binary classification dataset
58      dataset_filepath = os.path.join(
59          os.path.dirname(os.path.realpath(__file__)), 'diabetes.txt')
60      test_size = 0.33    # the percentage of samples in the dataset that will be
61      # randomly selected and assigned to the test set
62      n_labeled = 10      # number of samples that are initially labeled
63
64      # Load dataset
65      trn_ds, tst_ds, y_train, fully_labeled_trn_ds = \
66          split_train_test(dataset_filepath, test_size, n_labeled)
67      trn_ds2 = copy.deepcopy(trn_ds)
68      lbr = IdealLabeler(fully_labeled_trn_ds)
69
70      quota = len(y_train) - n_labeled    # number of samples to query
71
72      # Comparing UncertaintySampling strategy with RandomSampling.
73      # model is the base learner, e.g. LogisticRegression, SVM ... etc.
74      qs = UncertaintySampling(trn_ds, method='lc', model=LogisticRegression())
75      model = LogisticRegression()
76      E_in_1, E_out_1 = run(trn_ds, tst_ds, lbr, model, qs, quota)
77
78      qs2 = RandomSampling(trn_ds2)
79      model = LogisticRegression()
```

(continues on next page)

(continued from previous page)

```
    E_in_2, E_out_2 = run(trn_ds2, tst_ds, lbr, model, qs2, quota)

    # Plot the learning curve of UncertaintySampling to RandomSampling
    # The x-axis is the number of queries, and the y-axis is the corresponding
    # error rate.
    query_num = np.arange(1, quota + 1)
    plt.plot(query_num, E_in_1, 'b', label='qs Ein')
    plt.plot(query_num, E_in_2, 'r', label='random Ein')
    plt.plot(query_num, E_out_1, 'g', label='qs Eout')
    plt.plot(query_num, E_out_2, 'k', label='random Eout')
    plt.xlabel('Number of Queries')
    plt.ylabel('Error')
    plt.title('Experiment Result')
    plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05),
               fancybox=True, shadow=True, ncol=5)
    plt.show()


if __name__ == '__main__':
    main()
```

1.2.2 Interactive Digits Labeling

Example file: `examples/label_digits.py`

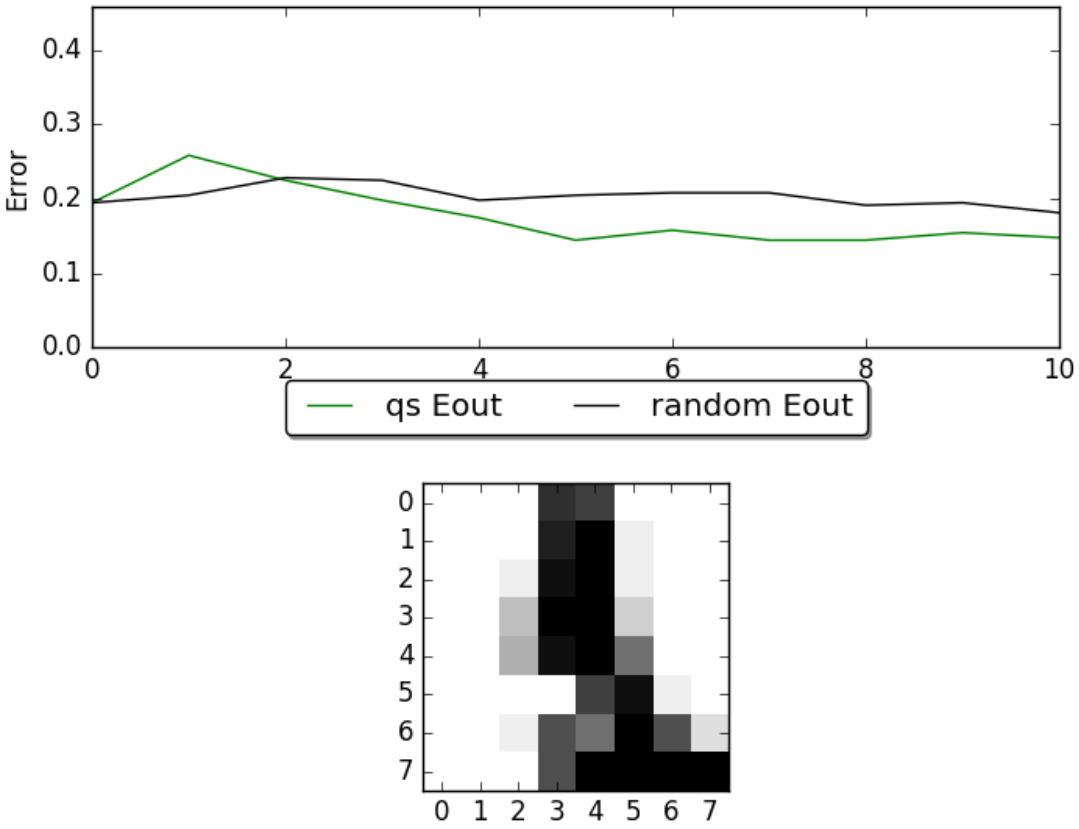
This example simiulates the use case where you want to humane to assign label to active learning algorithm selected samples. We uses the digits dataset provided by scikit-learn. Each time a sample is selected by active learning algorithm, the sample (a written digit) will be shown on the screen. The user will have to enter the corresponding digit to the command line to finish the labeling process.

The useage is roughly the same as the `plot.py` example.

```
    # Give each label its name (labels are from 0 to n_classes-1)
    lbr = InteractiveLabeler(label_name=[str(lbl) for lbl in range(n_classes)])

    for i in range(quota):
        ask_id = qs.make_query()
        print("asking sample from Uncertainty Sampling")
```

The difference is that the `labeler` is replaced by `InteractiveLabeler`, which opens the digit image for human labeler to see and receive the answer from command line.

Here are a snapshot of this example:



The figure on the top is the learning curve of ucertainty sampling and random sampling. X-axis is the number samples of queried, and Y-axis is the corresponding error rate. The figure on the button is the sample that human should assign label to.

Full source code:

```
#!/usr/bin/env python3
"""
This script simulates real world use of active learning algorithms. Which in the
start, there are only a small fraction of samples are labeled. During active
learing process active learning algorithm (QueryStrategy) will choose a sample
from unlabeled samples to ask the oracle to give this sample a label (Labeler).

In this example, ther dataset are from the digits dataset from sklearn. User
would have to label each sample choosed by QueryStrategy by hand. Human would
label each selected sample through InteractiveLabeler. Then we will compare the
performance of using UncertaintySampling and RandomSampling under
LogisticRegression.
"""

import copy

import numpy as np
import matplotlib.pyplot as plt
try:
```

(continues on next page)

(continued from previous page)

```
    from sklearn.model_selection import train_test_split
except ImportError:
    from sklearn.cross_validation import train_test_split

# libact classes
from libact.base.dataset import Dataset
from libact.models import LogisticRegression
from libact.query_strategies import UncertaintySampling, RandomSampling
from libact.labelers import InteractiveLabeler


def split_train_test(n_classes):
    from sklearn.datasets import load_digits

    n_labeled = 5
    digits = load_digits(n_class=n_classes)  # consider binary case
    X = digits.data
    y = digits.target
    print(np.shape(X))

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33)
    while len(np.unique(y_train[:n_labeled])) < n_classes:
        X_train, X_test, y_train, y_test = train_test_split(
            X, y, test_size=0.33)

    trn_ds = Dataset(X_train, np.concatenate(
        [y_train[:n_labeled], [None] * (len(y_train) - n_labeled)]))
    tst_ds = Dataset(X_test, y_test)

    return trn_ds, tst_ds, digits


def main():
    quota = 10  # ask human to label 10 samples
    n_classes = 5
    E_out1, E_out2 = [], []

    trn_ds, tst_ds, ds = split_train_test(n_classes)
    trn_ds2 = copy.deepcopy(trn_ds)

    qs = UncertaintySampling(trn_ds, method='lc', model=LogisticRegression())
    qs2 = RandomSampling(trn_ds2)

    model = LogisticRegression()

    fig = plt.figure()
    ax = fig.add_subplot(2, 1, 1)
    ax.set_xlabel('Number of Queries')
    ax.set_ylabel('Error')

    model.train(trn_ds)
    E_out1 = np.append(E_out1, 1 - model.score(tst_ds))
    model.train(trn_ds2)
    E_out2 = np.append(E_out2, 1 - model.score(tst_ds))

    query_num = np.arange(0, 1)
    p1, = ax.plot(query_num, E_out1, 'g', label='qs Eout')
```

(continues on next page)

(continued from previous page)

```
    p2, = ax.plot(query_num, E_out2, 'k', label='random Eout')
    plt.legend(loc='upper center', bbox_to_anchor=(0.5, -0.05), fancybox=True,
               shadow=True, ncol=5)
    plt.show(block=False)

    img_ax = fig.add_subplot(2, 1, 2)
    box = img_ax.get_position()
    img_ax.set_position([box.x0, box.y0 - box.height * 0.1, box.width,
                         box.height * 0.9])
    # Give each label its name (labels are from 0 to n_classes-1)
    lbr = InteractiveLabeler(label_name=[str(lbl) for lbl in range(n_classes)])

    for i in range(quota):
        ask_id = qs.make_query()
        print("asking sample from Uncertainty Sampling")
        # reshape the image to its width and height
        lb = lbr.label(trn_ds.data[ask_id][0].reshape(8, 8))
        trn_ds.update(ask_id, lb)
        model.train(trn_ds)
        E_out1 = np.append(E_out1, 1 - model.score(tst_ds))

        ask_id = qs2.make_query()
        print("asking sample from Random Sample")
        lb = lbr.label(trn_ds2.data[ask_id][0].reshape(8, 8))
        trn_ds2.update(ask_id, lb)
        model.train(trn_ds2)
        E_out2 = np.append(E_out2, 1 - model.score(tst_ds))

        ax.set_xlim((0, i + 1))
        ax.set_ylim((0, max(max(E_out1), max(E_out2)) + 0.2))
        query_num = np.arange(0, i + 2)
        p1.set_xdata(query_num)
        p1.set_ydata(E_out1)
        p2.set_xdata(query_num)
        p2.set_ydata(E_out2)

        plt.draw()

    input("Press any key to continue...")

if __name__ == '__main__':
    main()
```

1.2.3 Multilabel Query Strategies

Example file: `examples/multilabel_plot.py`

This example demonstrates the usage of *libact* in multilabel setting, which is the same under binary-class setting. This examples compares with the three multilabel active learning algorithms (Binary Minimization (BinMin), Maximal Loss Reduction with Maximal Confidence (MMC), Multilabel Active Learning With Auxiliary Learner (MLALAL). BinMin calculates the uncertainty of each label independently while MMC and MLALAL computes the uncertainty through evaluating the difference between predictions from two different multilabel classifiers. MMC has these two multilabel classifiers and the formula of evaluating the difference in prediction fixed. The multilabel classifiers it uses is binary relevance and stacked logistic regression. MLALAL is a more generalized version, we are able to freely assign multilabel classifiers and *libact* provides three different options for evaluating the difference in prediction (hamming

loss reduction, soft hamming loss reduction and, maximum margin reduction).

From the example we can see how these algorithms are assigned.

For BinMin, we only need a ContinuousModel for it to evaluate uncertainty.

```
qs6 = BinaryMinimization(trn_ds6, LogisticRegression())
```

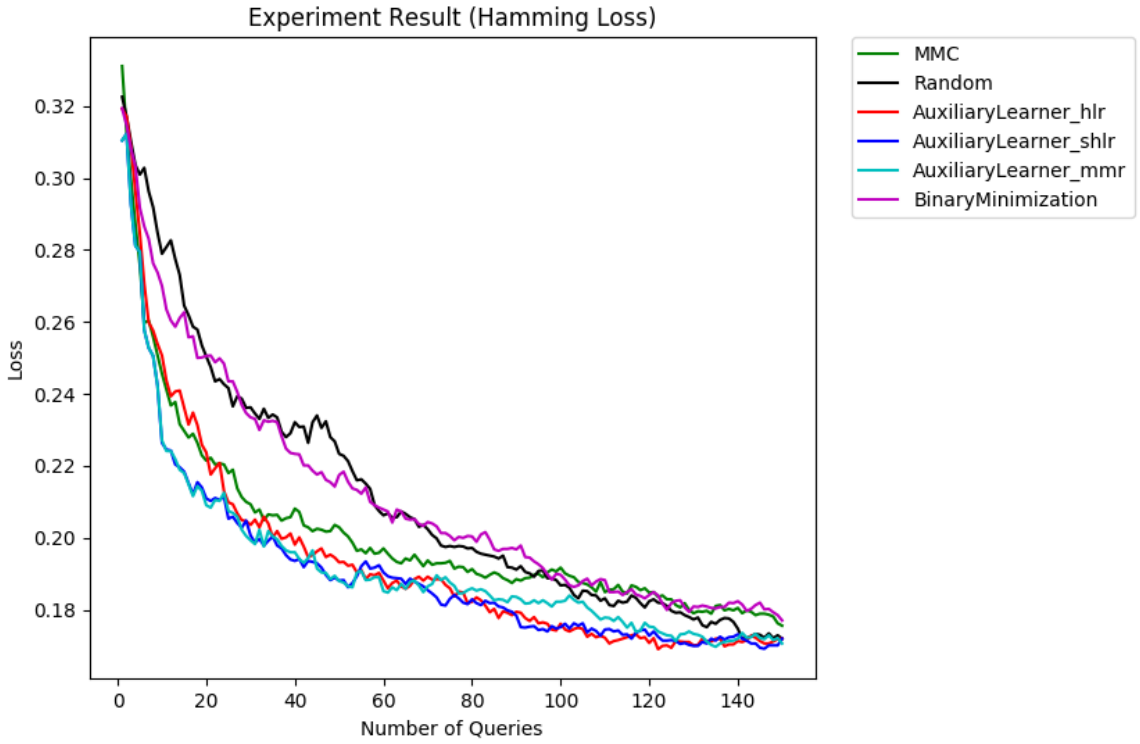
MMC on the other hand, it needs a base learner for its binary relevance.

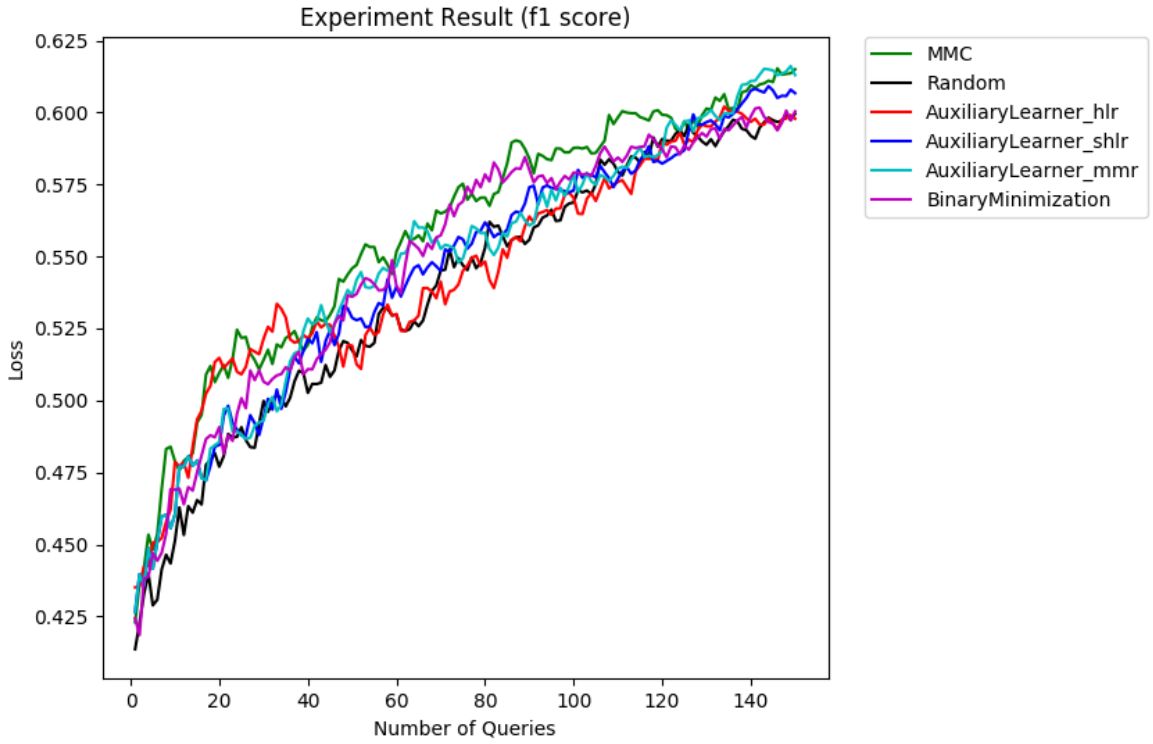
```
qs = MMC(trn_ds, br_base=LogisticRegression())
```

MLALAL need to assign two multilabel models. One serves as *major_learner*, and another serves as *auxiliary_learner*. The *major_leaner* should be the model to be use for final prediction and gives a binary output on each label. *auxiliary_learner* is only use to estimate the confident on each label, it should give a real value output (supports *pred_real* method).

```
qs3 = MultilabelWithAuxiliaryLearner(
    trn_ds3,
    BinaryRelevance(LogisticRegression()),
    BinaryRelevance(SVM()),
    criterion='hlr')
```

The results of this example on a artificial generated from *sklearn* is shown as follows:





1.3 Active Learning By Learning

Currently, most pool-based active learning algorithms are designed based on different human-designed philosophy, it is hard for user to decide which algorithm to use with a given problem. *Active Learning By Learning* (ALBL) algorithm is a meta active learn algorithm designed to solve this problem. ALBL considers multiple existing active learning algorithms and adaptively *learns* a querying strategy based on the performance of these algorithms.

ALBL's design is based on a well-known adaptive learning problem called multi-armed bandit. In the problem, K bandit machines and a budget of T iterations are given. Each time a bandit machine is pulled, the machine returns a reward that reflects the goodness of the machine. The multi-armed bandit problem aims at balancing between exploring each bandit machine and exploit the observed information in order to maximize the cumulative rewards after a seris of pulling decisions. The details can be found in the paper

Wei-Ning Hsu, and Hsuan-Tien Lin. "Active Learning by Learning." Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015.

Here is an example of how to declare a ALBL query_strategy object:

```
from libact.query_strategies import ActiveLearningByLearning
from libact.query_strategies import HintSVM
from libact.query_strategies import UncertaintySampling
from libact.models import LogisticRegression

qs = ActiveLearningByLearning(
         dataset, # Dataset object
         query_strategies=[
             UncertaintySampling(dataset, model=LogisticRegression(C=1.)),
```

(continues on next page)

(continued from previous page)

```
            UncertaintySampling(dataset, model=LogisticRegression(C=.01)),
            HintSVM(dataset)
            ],
        model=LogisticRegression()
        )
```

The `query_strategies` parameter is a list of `libact.query_strategies` object instances where each of their associated dataset must be the same `Dataset` instance. ALBL combines the result of these query strategies and generate its own suggestion of which sample to query. ALBL will adaptively *learn* from each of the decision it made, using the given supervised learning model in `model` parameter.

1.4 Cost Sensitive Active Learning

Most active learning algorithms are designed to deal with a specific miss-classification error. Though in the real-world applications, the cost for miss-classification varies. Cost-sensitive active learning algorithms allows the user to pass in the cost matrix as a parameter and select the data points that it thinks to perform the best on the given cost matrix.

Assume we have a total of K classes, **cost matrix** can be represented as a K*K matrix. The i-th row, j-th column represents the cost of the ground truth being i-th class and prediction as j-th class. The goal is to minimize the total cost.

libact provided the algorithm Active Learning with Cost-Embedding (ALCE) (libact.query_strategies.multiclass.ActiveLearningWithCostEmbedding) dedicated to solve this problem.

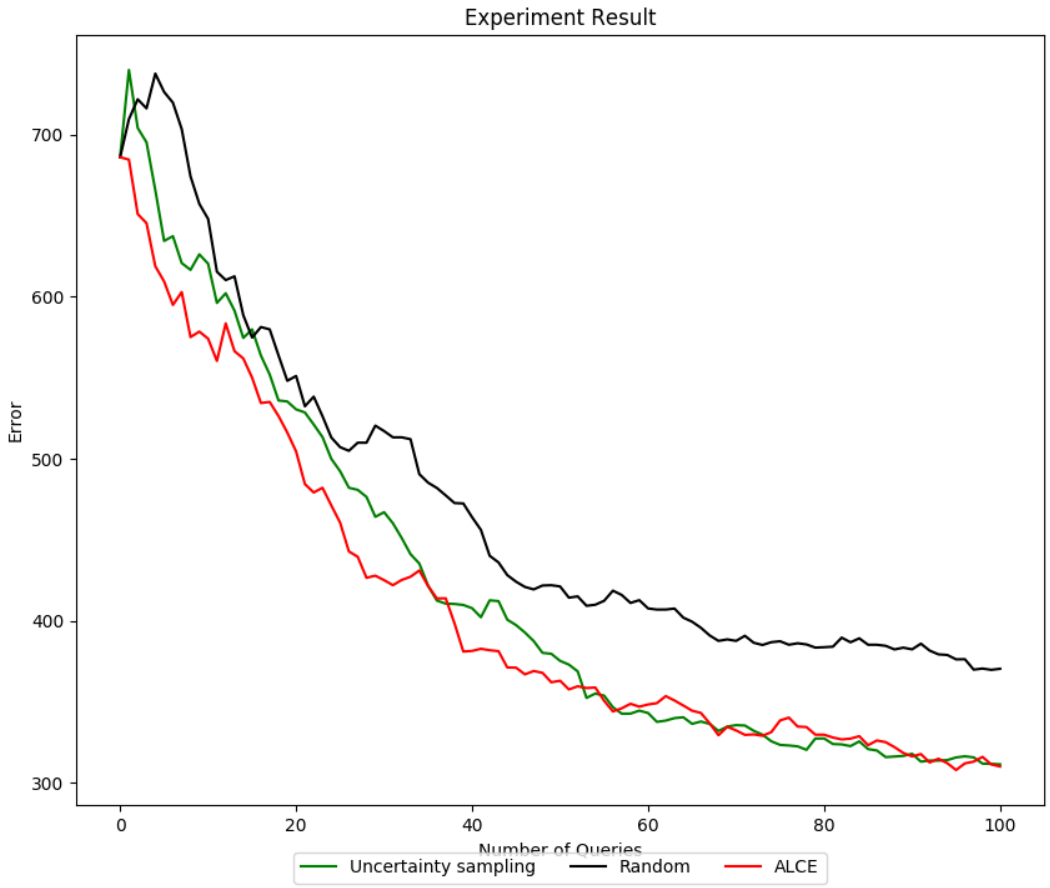
Example file: `examples/alce_plot.py`

The multi-class dataset to use is the *vehicle* dataset from mldata retrieved by sklearn (sklearn.datasets.fetch_mldata('vehicle')). The cost matrix is generated randomly.

```
cost_matrix = 2000. * np.random.rand(len(target), len(target))
```

The `target` variable is a list of different classes. The value `cost_matrix[i][j]` represent the cost of i-th class in `target` being predicted as j-th class in `target`.

In this example, we compared ALCE with Ucertainty Sampling and Random Sampling. The main difference in declaring an ALCE object is the cost_matrix should be passed in as a parameter (`ALCE(trn_ds3, cost_matrix, SVR())`). The result is shown as follows.



1.5 Develop with Libact

To develope active learning usage under *libact* framwork, you may implement your own oracle, active learning algorithm and machine learning algorithms.

1.5.1 Write your own models

To implement your own models, your model class should inherent from either *libact.base.interfaces.Model* or *libact.base.interfaces.ContinuousModel*. For regular model, there are three methods to be implmented: `train()`, `predict()`, and `score()`. For learning models that supports continuous output, method `predict_real()` should be implemented for `ContinuousModel`.

train

Method `train` takes in a `Dataset` object, which may include both labeled and unlabeled data. With supervised learning models, labeled data can be retrieved like this:

```
X, y = zip(*Dataset.get_labeled_entries())
```

`X, y` is the samples (shape=(n_samples, n_feature)) and labels (shape=(n_samples)).

You should train your model in this method like the `fit` method in *scikit-learn* model.

predict

This method should work like the `predict` method in *scikit-learn* model. Takes in the feature of each sample and output the label of the prediction for these samples.

score

This method should calculate the accuracy on a given dataset's labeled data.

predict_real

For models that can generate continuous predictions (for example, the distance to boundary).

Examples

Take a look at *libact.models.svm.SVM*, it serves as an interface of scikit-learn's SVC model. The train method is connected to scikit-learn's fit method and predict is connected to scikit-learn's predict. For the predict_real method, it represens the decision value to each label.

```
class SVM(ContinuousModel):

    """C-Support Vector Machine Classifier

    When decision_function_shape == 'ovr', we use OneVsRestClassifier(SVC) from
    sklearn.multiclass instead of the output from SVC directory since it is not
    exactly the implementation of One Vs Rest.

    References
    ----------
    http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
    """

    def __init__(self, *args, **kwargs):
        self.model = sklearn.svm.SVC(*args, **kwargs)
        if self.model.decision_function_shape == 'ovr':
            self.decision_function_shape = 'ovr'
            # sklearn's ovr isn't real ovr
            self.model = OneVsRestClassifier(self.model)

    def train(self, dataset, *args, **kwargs):
        return self.model.fit(*(dataset.format_sklearn() + args), **kwargs)

    def predict(self, feature, *args, **kwargs):
        return self.model.predict(feature, *args, **kwargs)

    def score(self, testing_dataset, *args, **kwargs):
        return self.model.score(*(testing_dataset.format_sklearn() + args),
```

(continues on next page)

(continued from previous page)

```
                                **kwargs)

    def predict_real(self, feature, *args, **kwargs):
        dvalue = self.model.decision_function(feature, *args, **kwargs)
        if len(np.shape(dvalue)) == 1:  # n_classes == 2
            return np.vstack((-dvalue, dvalue)).T
        else:
            if self.decision_function_shape != 'ovr':
                LOGGER.warn("SVM model support only 'ovr' for multiclass"
                            "predict_real.")
            return dvalue
```

1.5.2 Implement your active learning algorithm

You may implement your own active learning algorithm under QueryStrategy classes. QueryStrategy class should inherent from *libact.base.interfaces.QueryStrategy* and add the following into your __init__ method.

```
super(YourClassName, self).__init__(*args, **kwargs)
```

This would associate the given dataset with your query strategy and registers the update method under the associated dataset as a callback function.

The `update()` method should be used if the active learning algorithm wants to change its internal state after the dataset is updated with newly retrieved label. Take ALBL's `update()` method as example:

```
    @inherit_docstring_from(QueryStrategy)
    def update(self, entry_id, label):
        # Calculate the next query after updating the question asked with an
        # answer.
        ask_idx = self.unlabeled_invert_id_idx[entry_id]
        self.W.append(1. / self.query_dist[ask_idx])
        self.queried_hist_.append(entry_id)
```

`make_query()` is another method need to be implmented. It calculates which sample to query and outputs the entry id of that sample. Take the uncertainty sampling algorithm as example:

```
    def make_query(self, return_score=False):
        """Return the index of the sample to be queried and labeled and
        selection score of each sample. Read-only.

        No modification to the internal states.

        Returns
        -------
        ask_id : int
            The index of the next unlabeled sample to be queried and labeled.

        score : list of (index, score) tuple
            Selection score of unlabled entries, the larger the better.

        """
        dataset = self.dataset
        self.model.train(dataset)

        unlabeled_entry_ids, X_pool = zip(*dataset.get_unlabeled_entries())
```

(continues on next page)

(continued from previous page)

```
        if isinstance(self.model, ProbabilisticModel):
            dvalue = self.model.predict_proba(X_pool)
        elif isinstance(self.model, ContinuousModel):
            dvalue = self.model.predict_real(X_pool)

        if self.method == 'lc':  # least confident
            score = -np.max(dvalue, axis=1)

        elif self.method == 'sm':  # smallest margin
            if np.shape(dvalue)[1] > 2:
                # Find 2 largest decision values
                dvalue = -(np.partition(-dvalue, 2, axis=1)[:, :2])
            score = -np.abs(dvalue[:, 0] - dvalue[:, 1])

        elif self.method == 'entropy':
            score = np.sum(-dvalue * np.log(dvalue), axis=1)

        ask_id = np.argmax(score)

        if return_score:
            return unlabeled_entry_ids[ask_id], \
                   list(zip(unlabeled_entry_ids, score))
        else:
            return unlabeled_entry_ids[ask_id]
```

In uncertainty sampling, it asks the sample with the lowest decision value (the output from `predict_real()` of a `ContinuousModel`).

1.5.3 Write your Oracle

Different usage requires different ways of retrieving the label for an unlabeled sameple, therefore you may want to implement your own oracle for different condition To implement Labeler class you should inherent from *libact.base.interfaces.Labeler* and implment the `label()` function with how to retrieve the label of a given sample (feature).

Examples

We have provided two example labelers: `libact.labelers.IdealLabeler` and `libact.labelers.InteractiveLabeler`.

`IdealLabeler` is usually used for testing the performance of a active learning algorithm. You give it a fully-labeled dataset, simulating a oracle that know the true label of all samples. Its `label()` is simple searching through the given feature in the fully-labeled dataset and return the corresponding label.

```
class IdealLabeler(Labeler):

    """
    Provide the errorless/noiseless label to any feature vectors being queried.

    Parameters
    ----------
    dataset: Dataset object
        Dataset object with the ground-truth label for each sample.
```

(continues on next page)

(continued from previous page)

```
    """

    def __init__(self, dataset, **kwargs):
        X, y = zip(*dataset.get_entries())
        # make sure the input dataset is fully labeled
        assert (np.array(y) != np.array(None)).all()
        self.X = X
        self.y = y

    @inherit_docstring_from(Labeler)
    def label(self, feature):
        return self.y[np.where([np.array_equal(x, feature)
                                for x in self.X])[0][0]]
```

`InteractiveLabeler` can be used in the situation where you want to show your feature through image, let a human be the oracle and label the image interactively. To implement its `label()` method, it may include showing the feature through image using `matplotlib.pyplot.imshow()` and receive input through command line interface:

```
class InteractiveLabeler(Labeler):

    """Interactive Labeler

    InteractiveLabeler is a Labeler object that shows the feature through image
    using matplotlib and lets human label each feature through command line
    interface.

    Parameters
    ----------
    label_name: list
        Let the label space be from 0 to len(label_name)-1, this list
        corresponds to each label's name.

    """

    def __init__(self, **kwargs):
        self.label_name = kwargs.pop('label_name', None)

    @inherit_docstring_from(Labeler)
    def label(self, feature):
        plt.imshow(feature, cmap=plt.cm.gray_r, interpolation='nearest')
        plt.draw()

        banner = "Enter the associated label with the image: "

        if self.label_name is not None:
            banner += str(self.label_name) + ' '

        lbl = input(banner)

        while (self.label_name is not None) and (lbl not in self.label_name):
            print('Invalid label, please re-enter the associated label.')
            lbl = input(banner)

        return self.label_name.index(lbl)
```

1.6 API Reference

1.6.1 libact.base package

Submodules

libact.base.dataset module

The dataset class used in this package. Datasets consists of data used for training, represented by a list of (feature, label) tuples. May be exported in different formats for application on other libraries.

class libact.base.dataset.**Dataset**(*X=None*, *y=None*)

Bases: object

libact dataset object

Parameters

- **X** (*{array-like}, shape = (n_samples, n_features)*) – Feature of sample set.
- **y** (*list of {int, None}, shape = (n_samples)*) – The ground truth (label) for corresponding sample. Unlabeled data should be given a label None.

data

list, shape = (n_samples) – List of all sample feature and label tuple.

append(*feature*, *label=None*)

Add a (feature, label) entry into the dataset. A None label indicates an unlabeled entry.

Parameters

- **feature** (*{array-like}, shape = (n_features)*) – Feature of the sample to append to dataset.
- **label** (*{int, None}*) – Label of the sample to append to dataset. None if unlabeled.

Returns **entry_id** – entry_id for the appened sample.

Return type {int}

format_sklearn()

Returns dataset in (X, y) format for use in scikit-learn. Unlabeled entries are ignored.

Returns

- **X** (*numpy array, shape = (n_samples, n_features)*) – Sample feature set.
- **y** (*numpy array, shape = (n_samples)*) – Sample labels.

get_entries()

Return the list of all sample feature and ground truth tuple.

Returns **data** – List of all sample feature and label tuple.

Return type list, shape = (n_samples)

get_labeled_entries()

Returns list of labeled feature and their label

Returns **labeled_entries** – Labeled entries

Return type list of (feature, label) tuple

get_num_of_labels()
Number of distinct lebels in this object.

Returns n_labels

Return type int

get_unlabeled_entries()
Returns list of unlabeled features, along with their entry_ids

Returns unlabeled_entries – Labeled entries

Return type list of (entry_id, feature) tuple

labeled_uniform_sample(*sample_size*, *replace=True*)
Returns a Dataset object with labeled data only, which is resampled uniformly with given sample size. Parameter *replace* decides whether sampling with replacement or not.

Parameters **sample_size** –

len_labeled()
Number of labeled data entries in this object.

Returns n_samples

Return type int

len_unlabeled()
Number of unlabeled data entries in this object.

Returns n_samples

Return type int

on_update(*callback*)
Add callback function to call when dataset updated.

Parameters **callback** (*callable*) – The function to be called when dataset is updated.

update(*entry_id*, *new_label*)
Updates an entry with entry_id with the given label

Parameters

- **entry_id** (*int*) – entry id of the sample to update.
- **label** (*{int, None}*) – Label of the sample to be update.

libact.base.dataset.**import_libsvm_sparse**(*filename*)
Imports dataset file in libsvm sparse format

libact.base.dataset.**import_scipy_mat**(*filename*)

libact.base.interfaces module

Base interfaces for use in the package. The package works according to the interfaces defined below.

class libact.base.interfaces.**ContinuousModel**
Bases: *libact.base.interfaces.Model*

Classification Model with intermediate continuous output

A continuous classification model is able to output a real-valued vector for each features provided.

predict_real (*feature*, **args*, ***kwargs*)
Predict confidence scores for samples.

Returns the confidence score for each (sample, class) combination.

The larger the value for entry (sample=x, class=k) is, the more confident the model is about the sample x belonging to the class k.

Take Logistic Regression as example, the return value is the signed distance of that sample to the hyperplane.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose confidence scores are to be predicted.

Returns **X** – Each entry is the confidence scores per (sample, class) combination.

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

class libact.base.interfaces.**Labeler**
Bases: object

Label the queries made by QueryStrategies

Assign labels to the samples queried by QueryStrategies.

label (*feature*)
Return the class labels for the input feature array.

Parameters **feature** (*array-like, shape (n_features,)*) – The feature vector whose label is to queried.

Returns **label** – The class label of the queried feature.

Return type int

class libact.base.interfaces.**Model**
Bases: object

Classification Model

A Model returns a class-predicting function for future samples after trained on a training dataset.

predict (*feature*, **args*, ***kwargs*)
Predict the class labels for the input samples

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The unlabeled samples whose labels are to be predicted.

Returns **y_pred** – The class labels for samples in the feature array.

Return type array-like, shape (n_samples,)

score (*testing_dataset*, **args*, ***kwargs*)
Return the mean accuracy on the test dataset

Parameters **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train (*dataset*, **args*, ***kwargs*)
Train a model according to the given training dataset.

Parameters **dataset** (*Dataset object*) – The training dataset the model is to be trained on.

Returns **self** – Returns self.

Return type object

class libact.base.interfaces.**MultilabelModel**
Bases: *libact.base.interfaces.Model*

Multilabel Classification Model

A Model returns a multilabel-predicting function for future samples after trained on a training dataset.

class libact.base.interfaces.**ProbabilisticModel**
Bases: *libact.base.interfaces.ContinuousModel*

Classification Model with probability output

A probabilistic classification model is able to output a real-valued vector for each features provided.

predict_proba (*feature*, **args*, ***kwargs*)
Predict probability estimate for samples.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose probability estimation are to be predicted.

Returns **X** – Each entry is the prabablity estimate for each class.

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

predict_real (*feature*, **args*, ***kwargs*)
Predict confidence scores for samples.

Returns the confidence score for each (sample, class) combination.

The larger the value for entry (sample=x, class=k) is, the more confident the model is about the sample x belonging to the class k.

Take Logistic Regression as example, the return value is the signed distance of that sample to the hyperplane.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose confidence scores are to be predicted.

Returns **X** – Each entry is the confidence scores per (sample, class) combination.

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

class libact.base.interfaces.**QueryStrategy** (*dataset*, ***kwargs*)
Bases: object

Pool-based query strategy

A QueryStrategy advices on which unlabeled data to be queried next given a pool of labeled and unlabeled data.

dataset
The Dataset object that is associated with this QueryStrategy.

make_query ()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

update (*entry_id*, *label*)
Update the internal states of the QueryStrategy after each queried sample being labeled.

Parameters

- **entry_id** (*int*) – The index of the newly labeled sample.
- **label** (*float*) – The label of the queried sample.

Module contents

1.6.2 libact.labelers package

Submodules

libact.labelers.ideal_labeler module

Ideal/Noiseless labeler that returns true label

class libact.labelers.ideal_labeler.**IdealLabeler** (*dataset*, ***kwargs*)
Bases: *libact.base.interfaces.Labeler*

Provide the errorless/noiseless label to any feature vectors being queried.

Parameters **dataset** (*Dataset object*) – Dataset object with the ground-truth label for each sample.

label (*feature*)
Return the class labels for the input feature array.

Parameters **feature** (*array-like, shape (n_features,)*) – The feature vector whose label is to queried.

Returns **label** – The class label of the queried feature.

Return type int

libact.labelers.interactive_labeler module

Interactive Labeler

This module includes an InteractiveLabeler.

class libact.labelers.interactive_labeler.**InteractiveLabeler** (***kwargs*)
Bases: *libact.base.interfaces.Labeler*

Interactive Labeler

InteractiveLabeler is a Labeler object that shows the feature through image using matplotlib and lets human label each feature through command line interface.

Parameters **label_name** (*list*) – Let the label space be from 0 to len(label_name)-1, this list corresponds to each label's name.

label (*feature*)
Return the class labels for the input feature array.

Parameters **feature** (*array-like, shape (n_features,)*) – The feature vector whose label is to queried.

Returns **label** – The class label of the queried feature.

Return type int

Module contents

Concrete labeler classes.

1.6.3 libact.models package

Submodules

libact.models.multilabel package

Submodules

libact.models.multilabel.binary_relevance module

This module contains implementation of binary relevance for multi-label classification problems

class libact.models.multilabel.binary_relevance.**BinaryRelevance**(*base_clf*, *n_jobs=1*)

Bases: *libact.base.interfaces.MultilabelModel*

Binary Relevance

base_clf [*libact.models* object instances] If wanting to use predict_proba, base_clf are required to support predict_proba method.

n_jobs [int, optional, default: 1] The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

References

predict(*X*)

Predict labels.

Parameters **X** (*array-like, shape=(n_samples, n_features)*) – Feature vector.

Returns **pred** – Predicted labels of given feature vector.

Return type numpy array, shape=(n_samples, n_labels)

predict_proba(*X*)

Predict the probability of being 1 for each label.

Parameters **X** (*array-like, shape=(n_samples, n_features)*) – Feature vector.

Returns **pred** – Predicted probability of each label.

Return type numpy array, shape=(n_samples, n_labels)

predict_real(*X*)

Predict the probability of being 1 for each label.

Parameters **X** (*array-like, shape=(n_samples, n_features)*) – Feature vector.

Returns **pred** – Predicted probability of each label.

Return type numpy array, shape=(n_samples, n_labels)

score(*testing_dataset*, *criterion='hamming'*)
Return the mean accuracy on the test dataset

Parameters

- **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.
- **criterion** (*['hamming', 'f1']*) – instance-wise criterion.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train(*dataset*)
Train model with given feature.

Parameters

- **X** (*array-like, shape=(n_samples, n_features)*) – Train feature vector.
- **Y** (*array-like, shape=(n_samples, n_labels)*) – Target labels.

clfs_
list of *libact.models* object instances – Classifier instances.

Returns **self** – Retuen self.

Return type object

libact.models.multilabel.dummy_clf module

This module provides a dummy classifier, since in multi-label active learning problem, it is common to see label being all zero in training set. We will let this classifier handles this condition.

class libact.models.multilabel.dummy_clf.**DummyClf**
Bases: object

This classifier handles training sets with only 0s or 1s to unify the interface.

fit(*X*, *y*)

predict(*X*)

predict_proba(*X*)

predict_real(*X*)

train(*dataset*)

Module contents

Concrete model classes.

libact.models.sklearn_adapter module

scikit-learn classifier adapter

class libact.models.sklearn_adapter.**SklearnAdapter**(*clf*)

Bases: *libact.base.interfaces.Model*

Implementation of the scikit-learn classifier to libact model interface.

Parameters **clf** (*scikit-learn classifier object instance*) – The classifier object that is intended to be use with libact

---

Examples

Here is an example of using SklearnAdapter to classify the iris dataset:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

from libact.base.dataset import Dataset
from libact.models import SklearnAdapter

iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

adapter = SklearnAdapter(LogisticRegression(random_state=1126))

adapter.train(Dataset(X_train, y_train))
adapter.predict(X_test)
```

---

predict(*feature*, **args*, ***kwargs*)

Predict the class labels for the input samples

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The unlabeled samples whose labels are to be predicted.

Returns **y_pred** – The class labels for samples in the feature array.

Return type array-like, shape (n_samples,)

score(*testing_dataset*, **args*, ***kwargs*)

Return the mean accuracy on the test dataset

Parameters **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train(*dataset*, **args*, ***kwargs*)

Train a model according to the given training dataset.

Parameters **dataset** (*Dataset object*) – The training dataset the model is to be trained on.

Returns **self** – Returns self.

Return type object

class libact.models.sklearn_adapter.**SklearnProbaAdapter**(*clf*)

Bases: *libact.base.interfaces.ProbabilisticModel*

Implementation of the scikit-learn classifier to libact model interface. It should support predict_proba method and predict_real is default to return predict_proba.

Parameters **clf** (*scikit-learn classifier object instance*) – The classifier object that is intended to be use with libact

---

Examples

Here is an example of using SklearnAdapter to classify the iris dataset:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

from libact.base.dataset import Dataset
from libact.models import SklearnProbaAdapter

iris = datasets.load_iris()
X = iris.data
y = iris.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

adapter = SklearnProbaAdapter(LogisticRegression(random_state=1126))

adapter.train(Dataset(X_train, y_train))
adapter.predict(X_test)
adapter.predict_proba(X_test)
```

---

predict (*feature*, **args*, ***kwargs*)

Predict the class labels for the input samples

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The unlabeled samples whose labels are to be predicted.

Returns **y_pred** – The class labels for samples in the feature array.

Return type array-like, shape (n_samples,)

predict_proba (*feature*, **args*, ***kwargs*)

Predict probability estimate for samples.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose probability estimation are to be predicted.

Returns **X** – Each entry is the prabablity estimate for each class.

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

predict_real (*feature*, **args*, ***kwargs*)

Predict confidence scores for samples.

Returns the confidence score for each (sample, class) combination.

The larger the value for entry (sample=x, class=k) is, the more confident the model is about the sample x belonging to the class k.

Take Logistic Regression as example, the return value is the signed distance of that sample to the hyperplane.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose confidence scores are to be predicted.

Returns **X** – Each entry is the confidence scores per (sample, class) combination.

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

score(*testing_dataset*, **args*, ***kwargs*)
Return the mean accuracy on the test dataset

Parameters **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train(*dataset*, **args*, ***kwargs*)
Train a model according to the given training dataset.

Parameters **dataset** (*Dataset object*) – The training dataset the model is to be trained on.

Returns **self** – Returns self.

Return type object

libact.models.logistic_regression module

This module includes a class for interfacing scikit-learn's logistic regression model.

class libact.models.logistic_regression.**LogisticRegression**(**args*, ***kwargs*)
Bases: *libact.base.interfaces.ProbabilisticModel*

Logistic Regression Classifier

References

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

---

predict(*feature*, **args*, ***kwargs*)
Predict the class labels for the input samples

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The unlabeled samples whose labels are to be predicted.

Returns **y_pred** – The class labels for samples in the feature array.

Return type array-like, shape (n_samples,)

predict_proba(*feature*, **args*, ***kwargs*)
Predict probability estimate for samples.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose probability estimation are to be predicted.

Returns **X** – Each entry is the prabablity estimate for each class.

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

predict_real(*feature*, **args*, ***kwargs*)
Predict confidence scores for samples.

Returns the confidence score for each (sample, class) combination.

The larger the value for entry (sample=x, class=k) is, the more confident the model is about the sample x belonging to the class k.

Take Logistic Regression as example, the return value is the signed distance of that sample to the hyperplane.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose confidence scores are to be predicted.

Returns **X** – Each entry is the confidence scores per (sample, class) combination.

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

score(*testing_dataset*, **args*, ***kwargs*)
Return the mean accuracy on the test dataset

Parameters **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train(*dataset*, **args*, ***kwargs*)
Train a model according to the given training dataset.

Parameters **dataset** (*Dataset object*) – The training dataset the model is to be trained on.

Returns **self** – Returns self.

Return type object

libact.models.perceptron module

This module includes a class for interfacing scikit-learn's perceptron model.

class libact.models.perceptron.**Perceptron**(**args*, ***kwargs*)
Bases: *libact.base.interfaces.Model*

A interface for scikit-learn's perceptron model

References

http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Perceptron.html

---

predict(*feature*, **args*, ***kwargs*)
Predict the class labels for the input samples

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The unlabeled samples whose labels are to be predicted.

Returns **y_pred** – The class labels for samples in the feature array.

Return type array-like, shape (n_samples,)

score(*testing_dataset*, **args*, ***kwargs*)
Return the mean accuracy on the test dataset

Parameters **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train(*dataset*, **args*, ***kwargs*)
Train a model according to the given training dataset.

Parameters **dataset** (*Dataset object*) – The training dataset the model is to be trained on.

Returns **self** – Returns self.

Return type object

libact.models.svm module

SVM

An interface for scikit-learn's C-Support Vector Classifier model.

class libact.models.svm.**SVM**(**args*, ***kwargs*)
Bases: *libact.base.interfaces.ContinuousModel*

C-Support Vector Machine Classifier

When decision_function_shape == 'ovr', we use OneVsRestClassifier(SVC) from sklearn.multiclass instead of the output from SVC directory since it is not exactly the implementation of One Vs Rest.

References

http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

predict(*feature*, **args*, ***kwargs*)
Predict the class labels for the input samples

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The unlabeled samples whose labels are to be predicted.

Returns **y_pred** – The class labels for samples in the feature array.

Return type array-like, shape (n_samples,)

predict_real(*feature*, **args*, ***kwargs*)
Predict confidence scores for samples.

Returns the confidence score for each (sample, class) combination.

The larger the value for entry (sample=x, class=k) is, the more confident the model is about the sample x belonging to the class k.

Take Logistic Regression as example, the return value is the signed distance of that sample to the hyperplane.

Parameters **feature** (*array-like, shape (n_samples, n_features)*) – The samples whose confidence scores are to be predicted.

Returns **X** – Each entry is the confidence scores per (sample, class) combination.

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

score(*testing_dataset*, **args*, ***kwargs*)
Return the mean accuracy on the test dataset

Parameters **testing_dataset** (*Dataset object*) – The testing dataset used to measure the perforance of the trained model.

Returns **score** – Mean accuracy of self.predict(X) wrt. y.

Return type float

train(*dataset*, **args*, ***kwargs*)
Train a model according to the given training dataset.

Parameters **dataset** (*Dataset object*) – The training dataset the model is to be trained on.

Returns **self** – Returns self.

Return type object

Module contents

Concrete model classes.

1.6.4 libact.query_strategies package

Submodules

libact.query_strategies.multiclass package

Submodules

libact.query_strategies.multiclass.active_learning_with_cost_embedding module

Active Learning with Cost Embedding (ALCE)

class libact.query_strategies.multiclass.active_learning_with_cost_embedding.**ActiveLearning**

Bases: *libact.base.interfaces.QueryStrategy*

Active Learning with Cost Embedding (ALCE)

Cost sensitive multi-class algorithm. Assume each class has at least one sample in the labeled pool.

Parameters

- **cost_matrix** (*array-like, shape=(n_classes, n_classes)*) – The ith row, jth column represents the cost of the ground truth being ith class and prediction as jth class.
- **mds_params** (*dict, optional*) – http://scikit-learn.org/stable/modules/generated/sklearn.manifold.MDS.html
- **nn_params** (*dict, optional*) – http://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html

- **embed_dim** (*int, optional (default: None)*) – if is None, embed_dim = n_classes
- **base_regressor** (*sklearn regressor*) –
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

nn_
sklearn.neighbors.NearestNeighbors object instance

Examples

Here is an example of declaring a ActiveLearningWithCostEmbedding query_strategy object:

```
import numpy as np
from sklearn.svm import SVR

from libact.query_strategies.multiclass import ActiveLearningWithCostEmbedding as␣
↪ALCE

cost_matrix = 2000. * np.random.rand(n_classes, n_classes)
qs3 = ALCE(dataset, cost_matrix, SVR())
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.uncertainty_sampling module

Uncertainty Sampling

This module contains a class that implements two of the most well-known uncertainty sampling query strategies: the least confidence method and the smallest margin method (margin sampling).

class libact.query_strategies.uncertainty_sampling.**UncertaintySampling**(**args*, ***kwargs*)

Bases: *libact.base.interfaces.QueryStrategy*

Uncertainty Sampling

This class implements Uncertainty Sampling active learning algorithm *[1]*.

Parameters

- **model** (*libact.base.interfaces.ContinuousModel* or *libact.base.interfaces.ProbabilisticModel* object instance) – The base model used for training.

- **method** (*{'lc', 'sm', 'entropy'}, optional (default='lc')*) – least confidence (lc), it queries the instance whose posterior probability of being positive is nearest 0.5 (for binary classification); smallest margin (sm), it queries the instance whose posterior probability gap between the most and the second probable labels is minimal; entropy, requires *libact.base.interfaces.ProbabilisticModel* to be passed in as model parameter;

model
libact.base.interfaces.ContinuousModel or *libact.base.interfaces.ProbabilisticModel* object instance – The model trained in last query.

Examples

Here is an example of declaring a UncertaintySampling query_strategy object:

```
from libact.query_strategies import UncertaintySampling
from libact.models import LogisticRegression

qs = UncertaintySampling(
         dataset, # Dataset object
         model=LogisticRegression(C=0.1)
     )
```

Note that the model given in the `model` parameter must be a `ContinuousModel` which supports predict_real method.

References

make_query(*return_score=False*)
Return the index of the sample to be queried and labeled and selection score of each sample. Read-only.

No modification to the internal states.

Returns

- **ask_id** (*int*) – The index of the next unlabeled sample to be queried and labeled.
- **score** (*list of (index, score) tuple*) – Selection score of unlabled entries, the larger the better.

libact.query_strategies.multiclass.expected_error_reduction module

Expected Error Reduction

class libact.query_strategies.multiclass.expected_error_reduction.**EER**(*dataset, model=None, loss='log', random_state=None*)

Bases: *libact.base.interfaces.QueryStrategy*

Expected Error Reduction(EER)

This class implements EER active learning algorithm *[1]*.

Parameters

- **model** (*libact.base.interfaces.ProbabilisticModel* object instance) – The base model used for training.
- **loss** (*{'01', 'log'}, optional (default='log')*) – The loss function expected to reduce
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

model
libact.base.interfaces.ProbabilisticModel object instance – The model trained in last query.

Examples

Here is an example of declaring a UncertaintySampling query_strategy object:

```
from libact.query_strategies import EER
from libact.models import LogisticRegression

qs = EER(dataset, model=LogisticRegression(C=0.1))
```

Note that the model given in the `model` parameter must be a `ContinuousModel` which supports predict_real method.

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.multiclass.hierarchical_sampling module

Hierarchical Sampling for Active Learning (HS)

This module contains a class that implements Hierarchical Sampling for Active Learning (HS).

class libact.query_strategies.multiclass.hierarchical_sampling.**HierarchicalSampling**(*dataset*, *classes*, *active_selec*, *subsample_qs=N*, *random_stat*

Bases: *libact.base.interfaces.QueryStrategy*

Hierarchical Sampling for Active Learning (HS)

HS is an active learning scheme that exploits cluster structure in data. The original C++ implementation by the authors can be found at: http://www.cs.columbia.edu/~djhsu/code/HS.tar.gz

Parameters

- **classes** (*list*) – List of distinct classes in data.
- **active_selecting** (*{True, False}, optional (default=True)*) – False (random selecting): sample weight of a pruning is its number of unsean leaves. True (active selecting): sample weight of a pruning is its weighted error bound.
- **subsample_qs** ({libact.base.interfaces.query_strategies, None}, optional (default=None)) – Subsample query strategy used to sample a node in the selected pruning. RandomSampling is used if None.
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

m
int – number of nodes

classes
list – List of distinct classes in data.

n
int – number of leaf nodes

num_class
int – number of classes

parent
np.array instance, shape = (m) – parent indices

left_child
np.array instance, shape = (m) – left child indices

right_child
np.array instance, shape = (m) – right child indices

size
np.array instance, shape = (m) – number of leaves in subtree

depth
np.array instance, shape = (m) – maximum depth in subtree

count
np.array instance, shape = (m, num_class) – node class label counts

total
np.array instance, shape = (m) – total node class labels seen (total[i] = Sum_j count[i][j])

lower_bound
np.array instance, shape = (m, num_class) – upper bounds on true node class label counts

upper_bound
np.array instance, shape = (m, num_class) – lower bounds on true node class label counts

admissible
np.array instance, shape = (m, num_class) – flag indicating if (node,label) is admissible

best_label
np.array instance, shape = (m) – best admissible label

random_states_
np.random.RandomState instance – The random number generator using.

Examples

Here is an example of declaring a HierarchicalSampling query_strategy object:

```
from libact.query_strategies import UncertaintySampling
from libact.query_strategies.multiclass import HierarchicalSampling

sub_qs = UncertaintySampling(
    dataset, method='sm', model=SVM(decision_function_shape='ovr'))

qs = HierarchicalSampling(
         dataset, # Dataset object
         dataset.get_num_of_labels(),
         active_selecting=True,
         subsample_qs=sub_qs
     )
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

report_all_label()
Return the best label of the asked entry.

Returns **labels** – The best label of all samples.

Return type list of object, shape=(m)

report_entry_label(*entry_id*)
Return the best label of the asked entry.

Parameters **entry_id** (*int*) – The index of the sample to ask.

Returns **label** – The best label of the given sample.

Return type object

update(*entry_id*, *label*)
Update the internal states of the QueryStrategy after each queried sample being labeled.

Parameters

- **entry_id** (*int*) – The index of the newly labeled sample.
- **label** (*float*) – The label of the queried sample.

Module contents

libact.query_strategies.multilabel package

Submodules

libact.query_strategies.multilabel.adaptive_active_learning module

Adaptive active learning

class libact.query_strategies.multilabel.adaptive_active_learning.**AdaptiveActiveLearning**(*da*
ba
be
ta
*n_*
ra
da

Bases: *libact.base.interfaces.QueryStrategy*

Adaptive Active Learning

This approach combines Max Margin Uncertainty Sampling and Label Cardinality Inconsistency.

Parameters

- **base_clf** (*ContinuousModel object instance*) – The base learner for binary relavance.
- **betas** (*list of float, 0 <= beta <= 1, default: [0., 0.1, .., 0.9, 1.]*) – List of trade-off parameter that balances the relative importance degrees of the two measures.
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.
- **n_jobs** (*int, optional, default: 1*) – The number of jobs to use for the computation. If -1 all CPUs are used. If 1 is given, no parallel computing code is used at all, which is useful for debugging. For n_jobs below -1, (n_cpus + 1 + n_jobs) are used. Thus for n_jobs = -2, all CPUs but one are used.

Examples

Here is an example of declaring a MMC query_strategy object:

```
from libact.query_strategies.multilabel import AdaptiveActiveLearning
from sklearn.linear_model import LogisticRegression

qs = AdaptiveActiveLearning(
    dataset, # Dataset object
    base_clf=LogisticRegression()
)
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.multilabel.binary_minimization module

Binary Minimization

class libact.query_strategies.multilabel.binary_minimization.**BinaryMinimization**(*dataset*, *base_clf*, *random_state=None*)

Bases: *libact.base.interfaces.QueryStrategy*

Binary Version Space Minimization (BinMin)

Parameters

- **base_clf** (*ContinuousModel object instance*) – The base learner for binary relavance.
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

Examples

Here is an example of declaring a BinaryMinimization query_strategy object:

```
from libact.query_strategies.multilabel import BinaryMinimization
from sklearn.linear_model import LogisticRegression

qs = BinaryMinimization(
         dataset, # Dataset object
         br_base=LogisticRegression()
     )
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.multilabel.maximum_margin_reduction module

Maximum loss reduction with Maximal Confidence (MMC)

class libact.query_strategies.multilabel.maximum_margin_reduction.**MaximumLossReductionMaxi**

Bases: *libact.base.interfaces.QueryStrategy*

Maximum loss reduction with Maximal Confidence (MMC)

This algorithm is designed to use binary relavance with SVM as base model.

Parameters

- **base_learner** (*libact.query_strategies* object instance) – The base learner for binary relavance, should support predict_proba
- **br_base** (*ProbabilisticModel object instance*) – The base learner for the binary relevance in MMC. Should support predict_proba.
- **logreg_param** (*dict, optional (default={})*) – Setting the parameter for the logistic regression that are used to predict the number of labels for a given feature vector. Parameter detail please refer to: http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

logistic_regression_
libact.models.LogisticRegression object instance – The model used to predict the number of label in each instance. Should support multi-class classification.

Examples

Here is an example of declaring a MMC query_strategy object:

```
from libact.query_strategies.multilabel import MMC
from sklearn.linear_model import LogisticRegression

qs = MMC(
         dataset, # Dataset object
         br_base=LogisticRegression()
     )
```

References

---

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.multilabel.multilabel_with_auxiliary_learner module

Multi-label Active Learning with Auxiliary Learner

class libact.query_strategies.multilabel.multilabel_with_auxiliary_learner.**MultilabelWithAu**

Bases: *libact.base.interfaces.QueryStrategy*

Multi-label Active Learning with Auxiliary Learner

Parameters

- **major_learner** (*libact.base.interfaces.Model* object instance) – The major multilabel learner. This learner should be the model to be used to solve the problem.
- **auxiliary_learner** (libact.models.multilabel object instance) – The auxiliary multilabel learner. For criterion 'shlr' and 'mmr', it is required to support predict_real or predict_proba.
- **criterion** (*['hlr', 'shlr', 'mmr'], optional(default='hlr')*) – The criterion for estimating the difference between major_learner and auxiliary_learner. hlr, hamming loss reduction shlr, soft hamming loss reduction mmr, maximum margin reduction
- **b** (*float*) – parameter for criterion shlr. It sets the score to be clipped between [-b, b] to remove influence of extreme margin values.
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

Examples

Here is an example of declaring a multilabel with auxiliary learner query_strategy object:

```
from libact.query_strategies.multilabel import MultilabelWithAuxiliaryLearner
from libact.models.multilabel import BinaryRelevance
from libact.models import LogisticRegression, SVM

qs = MultilabelWithAuxiliaryLearner(
         dataset,
         major_learner=BinaryRelevance(LogisticRegression())
         auxiliary_learner=BinaryRelevance(SVM())
     )
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

Module contents

Concrete query strategy classes.

libact.query_strategies.active_learning_by_learning module

Active learning by learning (ALBL)

This module includes two classes. ActiveLearningByLearning is the main algorithm for ALBL and Exp4P is the multi-armed bandit algorithm which will be used in ALBL.

class libact.query_strategies.active_learning_by_learning.**ActiveLearningByLearning**(**args*, ***kwargs*)

Bases: *libact.base.interfaces.QueryStrategy*

Active Learning By Learning (ALBL) query strategy.

ALBL is an active learning algorithm that adaptively choose among existing query strategies to decide which data to make query. It utilizes Exp4.P, a multi-armed bandit algorithm to adaptively make such decision. More details of ALBL can refer to the work listed in the reference section.

Parameters

- **T** (*integer*) – Query budget, the maximal number of queries to be made.
- **query_strategies** (list of *libact.query_strategies*) –
- **instance** (*object*) – The active learning algorithms used in ALBL, which will be both the the arms in the multi-armed bandit algorithm Exp4.P. Note that these query_strategies should share the same dataset instance with ActiveLearningByLearning instance.
- **delta** (*float, optional (default=0.1)*) – Parameter for Exp4.P.
- **uniform_sampler** (*{True, False}, optional (default=True)*) – Determining whether to include uniform random sample as one of arms.
- **pmin** (float, 0<pmin< $\frac{1}{len(query\_strategies)}$,) – optional (default= $\frac{\sqrt{\log N}}{KT}$) Parameter for Exp4.P. The minimal probability for random selection of the arms (aka the underlying active learning algorithms). N = K = number of query_strategies, T is the number of query budgets.
- **model** (*libact.models* object instance) – The learning model used for the task.
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

query_strategies_
list of *libact.query_strategies* object instance – The active learning algorithm instances.

exp4p_
instance of Exp4P object – The multi-armed bandit instance.

queried_hist_
list of integer – A list of entry_id of the dataset which is queried in the past.

random_states_
np.random.RandomState instance – The random number generator using.

Examples

Here is an example of how to declare a ActiveLearningByLearning query_strategy object:

```
from libact.query_strategies import ActiveLearningByLearning
from libact.query_strategies import HintSVM
from libact.query_strategies import UncertaintySampling
from libact.models import LogisticRegression

qs = ActiveLearningByLearning(
     dataset, # Dataset object
     query_strategies=[
         UncertaintySampling(dataset, model=LogisticRegression(C=1.)),
         UncertaintySampling(dataset, model=LogisticRegression(C=.01)),
         HintSVM(dataset)
         ],
     model=LogisticRegression()
 )
```

The `query_strategies` parameter is a list of `libact.query_strategies` object instances where each of their associated dataset must be the same `Dataset` instance. ALBL combines the result of these query strategies and generate its own suggestion of which sample to query. ALBL will adaptively *learn* from each of the decision it made, using the given supervised learning model in `model` parameter to evaluate its IW-ACC.

References

calc_query()
Calculate the sampling query distribution

calc_reward_fn()
Calculate the reward value

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

update(*entry_id*, *label*)
Update the internal states of the QueryStrategy after each queried sample being labeled.

Parameters

- **entry_id** (*int*) – The index of the newly labeled sample.
- **label** (*float*) – The label of the queried sample.

class libact.query_strategies.active_learning_by_learning.**Exp4P**(**args*, ***kwargs*)

Bases: object

A multi-armed bandit algorithm Exp4.P.

For the Exp4.P used in ALBL, the number of arms (actions) and number of experts are equal to the number of active learning algorithms wanted to use. The arms (actions) are the active learning algorithms, where is inputed from parameter 'query_strategies'. There is no need for the input of experts, the advice of the kth expert are always equal e_k, where e_k is the kth column of the identity matrix.

Parameters

- **query_strategies** (*QueryStrategy instances*) – The active learning algorithms wanted to use, it is equivalent to actions or arms in original Exp4.P.
- **unlabeled_invert_id_idx** (*dict*) – A look up table for the correspondance of entry_id to the index of the unlabeled data.
- **delta** (*float, >0, optional (default=0.1)*) – A parameter.
- **pmin** (float, 0<pmin<1/len(query_strategies), optional (default= $\frac{\sqrt{log(N)}}{KT}$)) – The minimal probability for random selection of the arms (aka the unlabeled data), N = K = number of query_strategies, T is the maximum number of rounds.
- **T** (*int, optional (default=100)*) – The maximum number of rounds.
- **uniform_sampler** (*{True, False}, optional (default=Truee)*) – Determining whether to include uniform random sampler as one of the underlying active learning algorithms.

t
int – The current round this instance is at.

N
int – The number of arms (actions) in this exp4.p instance.

query_models_
list of *libact.query_strategies* object instance – The underlying active learning algorithm instances.

References

exp4p()
The generator which implements the main part of Exp4.P.

Parameters

- **reward** (*float*) – The reward value calculated from ALBL.
- **ask_id** (*integer*) – The entry_id of the sample point ALBL asked.
- **lbl** (*integer*) – The answer received from asking the entry_id ask_id.

Yields **q** (*array-like, shape = [K]*) – The query vector which tells ALBL what kind of distribution if should sample from the unlabeled pool.

next(*reward*, *ask_id*, *lbl*)
Taking the label and the reward value of last question and returns the next question to ask.

libact.query_strategies.hintsvm module

Hinted Support Vector Machine

This module contains a class that implements Hinted Support Vector Machine, an active learning algorithm.

Standalone hintsvm can be retrieved from https://github.com/yangarbiter/hintsvm

class libact.query_strategies.hintsvm.**HintSVM**(**args*, ***kwargs*)

Bases: *libact.base.interfaces.QueryStrategy*

Hinted Support Vector Machine

Hinted Support Vector Machine is an active learning algorithm within the hined sampling framework with an extended support vector machine.

Parameters

- **Cl** (*float, >0, optional (default=0.1)*) – The weight of the classification error on labeled pool.
- **Ch** (*float, >0, optional (default=0.1)*) – The weight of the hint error on hint pool.
- **p** (*float, >0 and <=1, optional (default=.5)*) – The probability to select an instance from unlabeld pool to hint pool.
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.
- **kernel** (*{'linear', 'poly', 'rbf', 'sigmoid'}, optional (default='linear')*) – linear: u'*v poly: (gamma*u'*v + coef0)^degree rbf: exp(-gamma*|u-v|^2) sigmoid: tanh(gamma*u'*v + coef0)
- **degree** (*int, optional (default=3)*) – Parameter for kernel function.
- **gamma** (*float, optional (default=0.1)*) – Parameter for kernel function.
- **coef0** (*float, optional (default=0.)*) – Parameter for kernel function.
- **tol** (*float, optional (default=1e-3)*) – Tolerance of termination criterion.
- **shrinking** (*{0, 1}, optional (default=1)*) – Whether to use the shrinking heuristics.
- **cache_size** (*float, optional (default=100.)*) – Set cache memory size in MB.
- **verbose** (*int, optional (default=0)*) – Set verbosity level for hintsvm solver.

random_states_

np.random.RandomState instance – The random number generator using.

Examples

Here is an example of declaring a HintSVM query_strategy object:

```
from libact.query_strategies import HintSVM

qs = HintSVM(
     dataset, # Dataset object
     Cl=0.01,
     p=0.8,
     )
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.query_by_committee module

Query by committee

This module contains a class that implements Query by committee active learning algorithm.

class libact.query_strategies.query_by_committee.**QueryByCommittee**(**args*, ***kwargs*)

Bases: *libact.base.interfaces.QueryStrategy*

Query by committee

Parameters

- **models** (list of *libact.models* instances or str) – This parameter accepts a list of initialized libact Model instances, or class names of libact Model classes to determine the models to be included in the committee to vote for each unlabeled instance.
- **disagreement** (*['vote', 'kl_divergence'], optional (default='vote')*) – Sets the method for measuring disagreement between models. 'vote' represents vote entropy. kl_divergence requires models being ProbabilisticModel
- **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

students
list, shape = (len(models)) – A list of the model instances used in this algorithm.

random_states_
np.random.RandomState instance – The random number generator using.

Examples

Here is an example of declaring a QueryByCommittee query_strategy object:

```
from libact.query_strategies import QueryByCommittee
from libact.models import LogisticRegression

qs = QueryByCommittee(
         dataset, # Dataset object
         models=[
             LogisticRegression(C=1.0),
             LogisticRegression(C=0.1),
         ],
     )
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

teach_students()
Train each model (student) with the labeled data using bootstrap aggregating (bagging).

update(*entry_id*, *label*)
Update the internal states of the QueryStrategy after each queried sample being labeled.

Parameters

- **entry_id** (*int*) – The index of the newly labeled sample.
- **label** (*float*) – The label of the queried sample.

libact.query_strategies.quire module

Active Learning by QUerying Informative and Representative Examples (QUIRE)

This module contains a class that implements an active learning algorithm (query strategy): QUIRE

class libact.query_strategies.quire.**QUIRE**(**args*, ***kwargs*)
Bases: *libact.base.interfaces.QueryStrategy*

Querying Informative and Representative Examples (QUIRE)

Query the most informative and representative examples where the metrics measuring and combining are done using min-max approach.

Parameters

- **lambda** (*float, optional (default=1.0)*) – A regularization parameter used in the regularization learning framework.
- **kernel** (*{'linear', 'poly', 'rbf', callable}, optional (default='rbf')*) – Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', or a callable. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape `(n_samples, n_samples)`.
- **degree** (*int, optional (default=3)*) – Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.
- **gamma** (*float, optional (default=1.)*) – Kernel coefficient for 'rbf', 'poly'.
- **coef0** (*float, optional (default=1.)*) – Independent term in kernel function. It is only significant in 'poly'.

Examples

Here is an example of declaring a QUIRE query_strategy object:

```
from libact.query_strategies import QUIRE

qs = QUIRE(
         dataset, # Dataset object
     )
```

References

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

update(*entry_id*, *label*)
Update the internal states of the QueryStrategy after each queried sample being labeled.

Parameters

- **entry_id** (*int*) – The index of the newly labeled sample.
- **label** (*float*) – The label of the queried sample.

libact.query_strategies.random_sampling module

Random Sampling

class libact.query_strategies.random_sampling.**RandomSampling**(*dataset*, ***kwargs*)
Bases: *libact.base.interfaces.QueryStrategy*

Random sampling

This class implements the random query strategy. A random entry from the unlabeled pool is returned for each query.

Parameters **random_state** (*{int, np.random.RandomState instance, None}, optional (default=None)*) – If int or None, random_state is passed as parameter to generate np.random.RandomState instance. if np.random.RandomState instance, random_state is the random number generate.

random_states_
np.random.RandomState instance – The random number generator using.

Examples

Here is an example of declaring a RandomSampling query_strategy object:

```
from libact.query_strategies import RandomSampling

qs = RandomSampling(
         dataset, # Dataset object
     )
```

make_query()
Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

libact.query_strategies.uncertainty_sampling module

Uncertainty Sampling

This module contains a class that implements two of the most well-known uncertainty sampling query strategies: the least confidence method and the smallest margin method (margin sampling).

class libact.query_strategies.uncertainty_sampling.**UncertaintySampling**(**args*, ***kwargs*)

Bases: *libact.base.interfaces.QueryStrategy*

Uncertainty Sampling

This class implements Uncertainty Sampling active learning algorithm *[1]*.

Parameters

- **model** (*libact.base.interfaces.ContinuousModel* or *libact.base.interfaces.ProbabilisticModel* object instance) – The base model used for training.
- **method** (*{'lc', 'sm', 'entropy'}, optional (default='lc')*) – least confidence (lc), it queries the instance whose posterior probability of being positive is nearest 0.5 (for binary classification); smallest margin (sm), it queries the instance whose posterior probability gap between the most and the second probable labels is minimal; entropy, requires *libact.base.interfaces.ProbabilisticModel* to be passed in as model parameter;

model
libact.base.interfaces.ContinuousModel or *libact.base.interfaces.ProbabilisticModel* object instance – The model trained in last query.

Examples

Here is an example of declaring a UncertaintySampling query_strategy object:

```
from libact.query_strategies import UncertaintySampling
from libact.models import LogisticRegression

qs = UncertaintySampling(
         dataset, # Dataset object
         model=LogisticRegression(C=0.1)
     )
```

Note that the model given in the `model` parameter must be a `ContinuousModel` which supports predict_real method.

References

---

make_query(*return_score=False*)

Return the index of the sample to be queried and labeled and selection score of each sample. Read-only.

No modification to the internal states.

Returns

- **ask_id** (*int*) – The index of the next unlabeled sample to be queried and labeled.
- **score** (*list of (index, score) tuple*) – Selection score of unlabled entries, the larger the better.

libact.query_strategies.variance_reduction module

Variance Reduction

class libact.query_strategies.variance_reduction.**VarianceReduction**(**args*, ***kwargs*)

Bases: *libact.base.interfaces.QueryStrategy*

Variance Reduction

This class implements Variance Reduction active learning algorithm *[1]*.

Parameters

- **model** (*{libact.model.LogisticRegression instance, 'LogisticRegression'}*) – The model used for variance reduction to evaluate the variance. Only Logistic regression are supported now.
- **sigma** (*float, >0, optional (default=100.0)*) – 1/sigma is added to the diagonal of the Fisher information matrix as a regularization term.
- **optimality** (*{'trace', 'determinant', 'eigenvalue'}, optional (default='trace')*) – The type of optimal design. The options are the trace, determinant, or maximum eigenvalue of the inverse Fisher information matrix. Only 'trace' are supported now.
- **n_jobs** (*int, optional (default=1)*) – The number of processors to estimate the expected variance.

References

make_query()

Return the index of the sample to be queried and labeled. Read-only.

No modification to the internal states.

Returns **ask_id** – The index of the next unlabeled sample to be queried and labeled.

Return type int

Module contents

Concrete query strategy classes.

- genindex
- modindex
- search

Bibliography

[1] Tsoumakas, Grigorios, Ioannis Katakis, and Ioannis Vlahavas. “Mining multi-label data.” Data mining and knowledge discovery handbook. Springer US, 2009. 667-685.

[1] Kuan-Hao, and Hsuan-Tien Lin. “A Novel Uncertainty Sampling Algorithm for Cost-sensitive Multiclass Active Learning”, In Proceedings of the IEEE International Conference on Data Mining (ICDM), 2016

[1] Settles, Burr. “Active learning literature survey.” University of Wisconsin, Madison 52.55-66 (2010): 11.

[1] Settles, Burr. “Active learning literature survey.” University of Wisconsin, Madison 52.55-66 (2010): 11.

[1] Sanjoy Dasgupta and Daniel Hsu. “Hierarchical sampling for active learning.” ICML 2008.

[1] Li, Xin, and Yuhong Guo. “Active Learning with Multi-Label SVM Classification.” IJCAI. 2013.

[1] Brinker, Klaus. “On active learning in multi-label classification.” From Data and Information Analysis to Knowledge Engineering. Springer Berlin Heidelberg, 2006. 206-213.

[1] Yang, Bishan, et al. “Effective multi-label active learning for text classification.” Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2009.

[1] Hung, Chen-Wei, and Hsuan-Tien Lin. “Multi-label Active Learning with Auxiliary Learner.” ACML. 2011.

[1] Wei-Ning Hsu, and Hsuan-Tien Lin. “Active Learning by Learning.” Twenty-Ninth AAAI Conference on Artificial Intelligence. 2015.

[1] Beygelzimer, Alina, et al. “Contextual bandit algorithms with supervised learning guarantees.” In Proceedings on the International Conference on Artificial Intelligence and Statistics (AISTATS), 2011u.

[1] Li, Chun-Liang, Chun-Sung Ferng, and Hsuan-Tien Lin. “Active Learning with Hinted Support Vector Machine.” ACML. 2012.

[2] Chun-Liang Li, Chun-Sung Ferng, and Hsuan-Tien Lin. Active learning using hint information. Neural Computation, 27(8):1738–1765, August 2015.

[1] Seung, H. Sebastian, Manfred Opper, and Haim Sompolinsky. “Query by committee.” Proceedings of the fifth annual workshop on Computational learning theory. ACM, 1992.

[1] S.-J. Huang, R. Jin, and Z.-H. Zhou. Active learning by querying informative and representative examples.

[1] Settles, Burr. “Active learning literature survey.” University of Wisconsin, Madison 52.55-66 (2010): 11.

[1] Schein, Andrew I., and Lyle H. Ungar. “Active learning for logistic regression: an evaluation.” Machine Learning 68.3 (2007): 235-265.

[2] Settles, Burr. “Active learning literature survey.” University of Wisconsin, Madison 52.55-66 (2010): 11.

Python Module Index

Index

L

M

T

U

V