modAL Documentation

Tivadar Danka

Oct 31, 2020

Overview

Welcome to the documentation for modAL!

modAL is an active learning framework for Python3, designed with *modularity, flexibility* and *extensibility* in mind. Built on top of scikit-learn, it allows you to rapidly create active learning workflows with nearly complete freedom. What is more, you can easily replace parts with your custom built solutions, allowing you to design novel algorithms with ease.

Currently supported active learning strategies are

- **uncertainty-based sampling:** *least confident* (Lewis and Catlett), *max margin* and *max entropy*
- **committee-based algorithms:** *vote entropy*, *consensus entropy* and *max disagreement* (Cohn et al.)
- **multilabel strategies:** *SVM binary minimum* (Brinker), *max loss*, *mean max loss*, (Li et al.) *MinConfidence*, *MeanConfidence*, *MinScore*, *MeanScore* (Esuli and Sebastiani)
- **expected error reduction:** *binary* and *log loss* (Roy and McCallum)
- **Bayesian optimization:** *probability of improvement*, *expected improvement* and *upper confidence bound* (Snoek et al.)
- **batch active learning:** *ranked batch-mode sampling* (Cardoso et al.)
- **information density framework** (McCallum and Nigam)
- **stream-based sampling** (Atlas et al.)
- **active regression** with *max standard deviance* sampling for Gaussian processes or ensemble regressors

CHAPTER 1

modAL in a nutshell

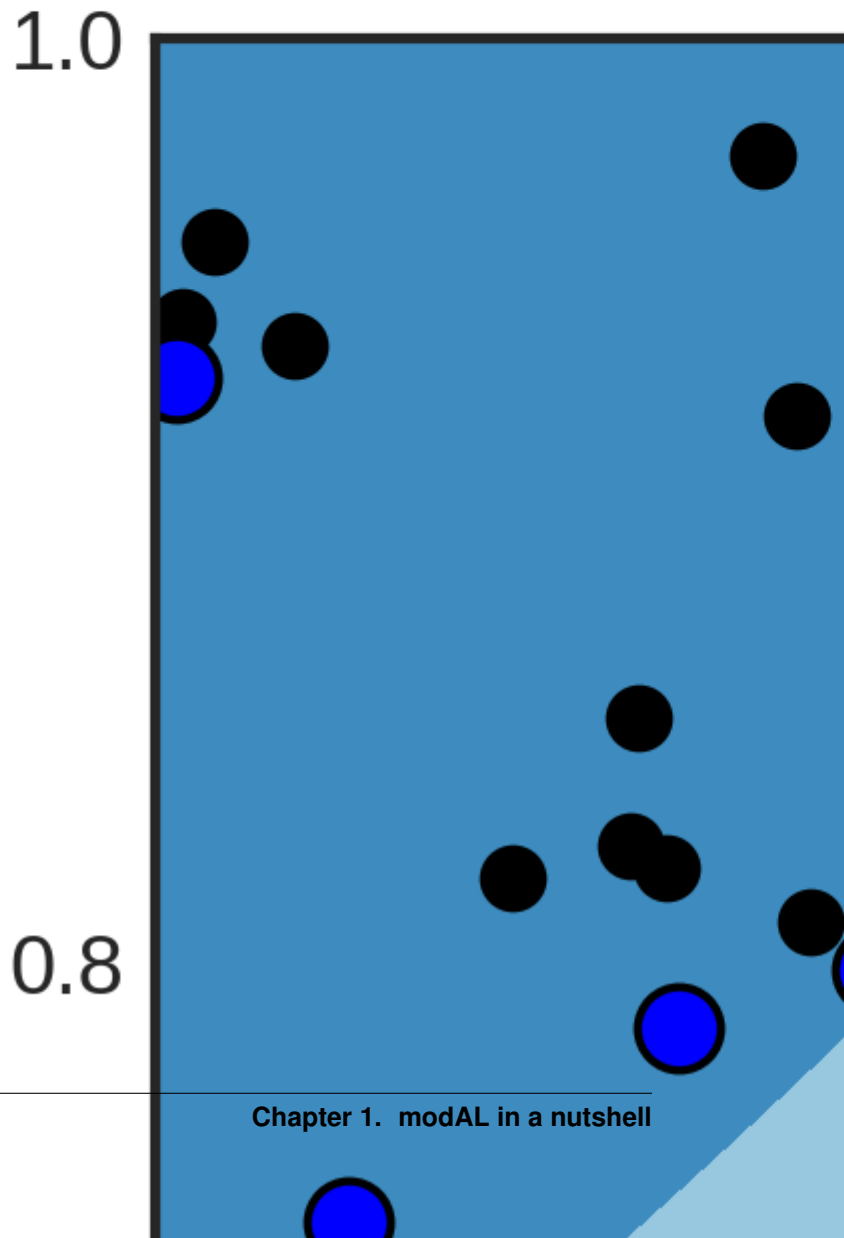
1.1 Introduction

modal: adjective, relating to structure as opposed to substance
(Merriam-Webster Dictionary)

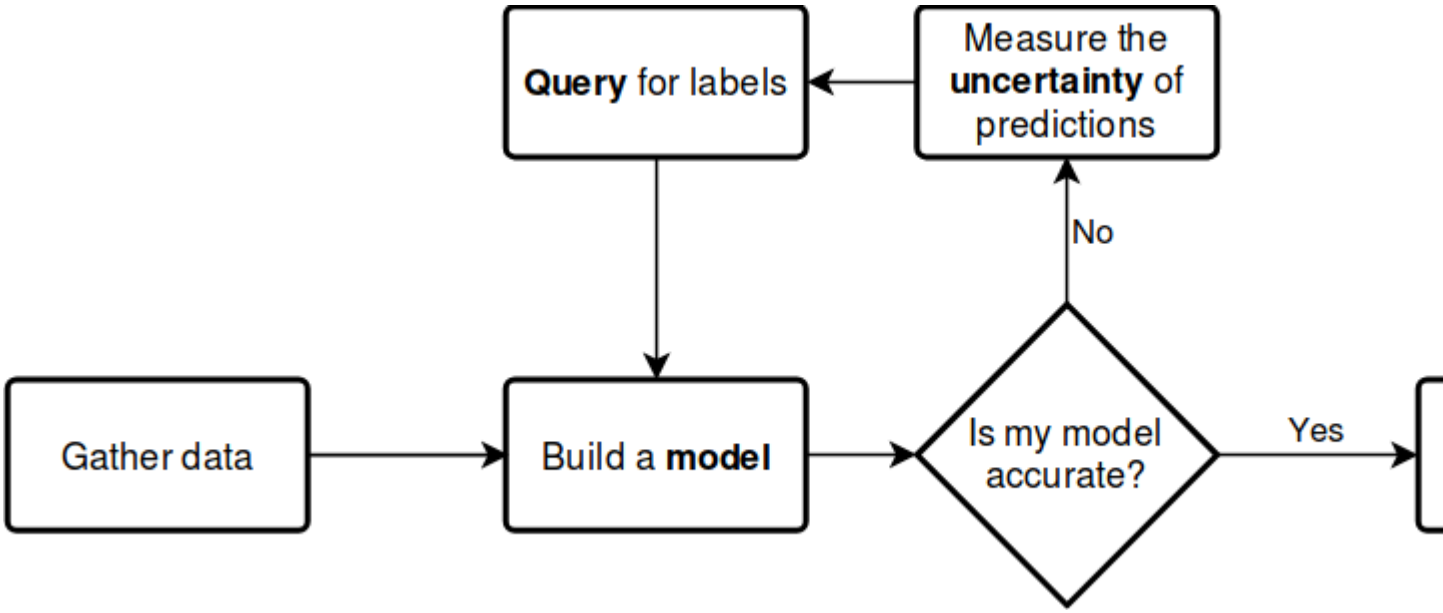
modAL is an active learning framework for Python3, designed with *modularity, flexibility* and *extensibility* in mind. Built on top of scikit-learn, it allows you to rapidly create active learning workflows with nearly complete freedom. What is more, you can easily replace parts with your custom built solutions, allowing you to design novel algorithms with ease.

1.2 Active learning from bird's-eye view

With the recent explosion of available data, you have can have millions of unlabelled examples with a high cost to obtain labels. For instance, when trying to predict the sentiment of tweets, obtaining a training set can require immense manual labour. But worry not, active learning comes to the rescue! In general, AL is a framework allowing you to increase classification performance by intelligently querying you to label the most informative instances. To give an example, suppose that you have the following data and classifier with shaded regions signifying the classification probability.


1.0
0.8

Suppose that you can query the label of an unlabelled instance, but it costs you a lot. Which one would you choose? By querying an instance in the uncertain region, surely you obtain more information than querying by random. Active learning gives you a set of tools to handle problems like this. In general, an active learning workflow looks like the following.



The key components of any workflow are the **model** you choose, the **uncertainty** measure you use and the **query** strategy you apply to request labels. With modAL, instead of choosing from a small set of built-in components, you have the freedom to seamlessly integrate scikit-learn or Keras models into your algorithm and easily tailor your custom query strategies and uncertainty measures.

1.3 modAL in action

Let's see what modAL can do for you!

1.3.1 From zero to one in a few lines of code

Active learning with a scikit-learn classifier, for instance RandomForestClassifier, can be as simple as the following.

```
from modAL.models import ActiveLearner
from sklearn.ensemble import RandomForestClassifier

# initializing the learner
learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    X_training=X_training, y_training=y_training
)

# query for labels
query_idx, query_inst = learner.query(X_pool)

# ...obtaining new labels from the Oracle...
```

(continues on next page)

(continued from previous page)

```
# supply label for queried instance
learner.teach(X_pool[query_idx], y_new)
```

1.3.2 Replacing parts quickly

If you would like to use different uncertainty measures and query strategies than the default uncertainty sampling, you can either replace them with several built-in strategies or you can design your own by following a few very simple design principles. For instance, replacing the default uncertainty measure to classification entropy looks the following.

```
from modAL.models import ActiveLearner
from modAL.uncertainty import entropy_sampling
from sklearn.ensemble import RandomForestClassifier

learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    query_strategy=entropy_sampling,
    X_training=X_training, y_training=y_training
)
```

1.3.3 Replacing parts with your own solutions

modAL was designed to make it easy for you to implement your own query strategy. For example, implementing and using a simple random sampling strategy is as easy as the following.

```
import numpy as np

def random_sampling(classifier, X_pool):
    n_samples = len(X_pool)
    query_idx = np.random.choice(range(n_samples))
    return query_idx, X_pool[query_idx]

learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    query_strategy=random_sampling,
    X_training=X_training, y_training=y_training
)
```

For more details on how to implement your custom strategies, visit the page Extending modAL!

1.3.4 An example with active regression

To see modAL in *real* action, let's consider an active regression problem with Gaussian processes! In this example, we shall try to learn the *noisy sine* function:

```
import numpy as np

X = np.random.choice(np.linspace(0, 20, 10000), size=200, replace=False).reshape(-1,␣
↪1)
y = np.sin(X) + np.random.normal(scale=0.3, size=X.shape)
```

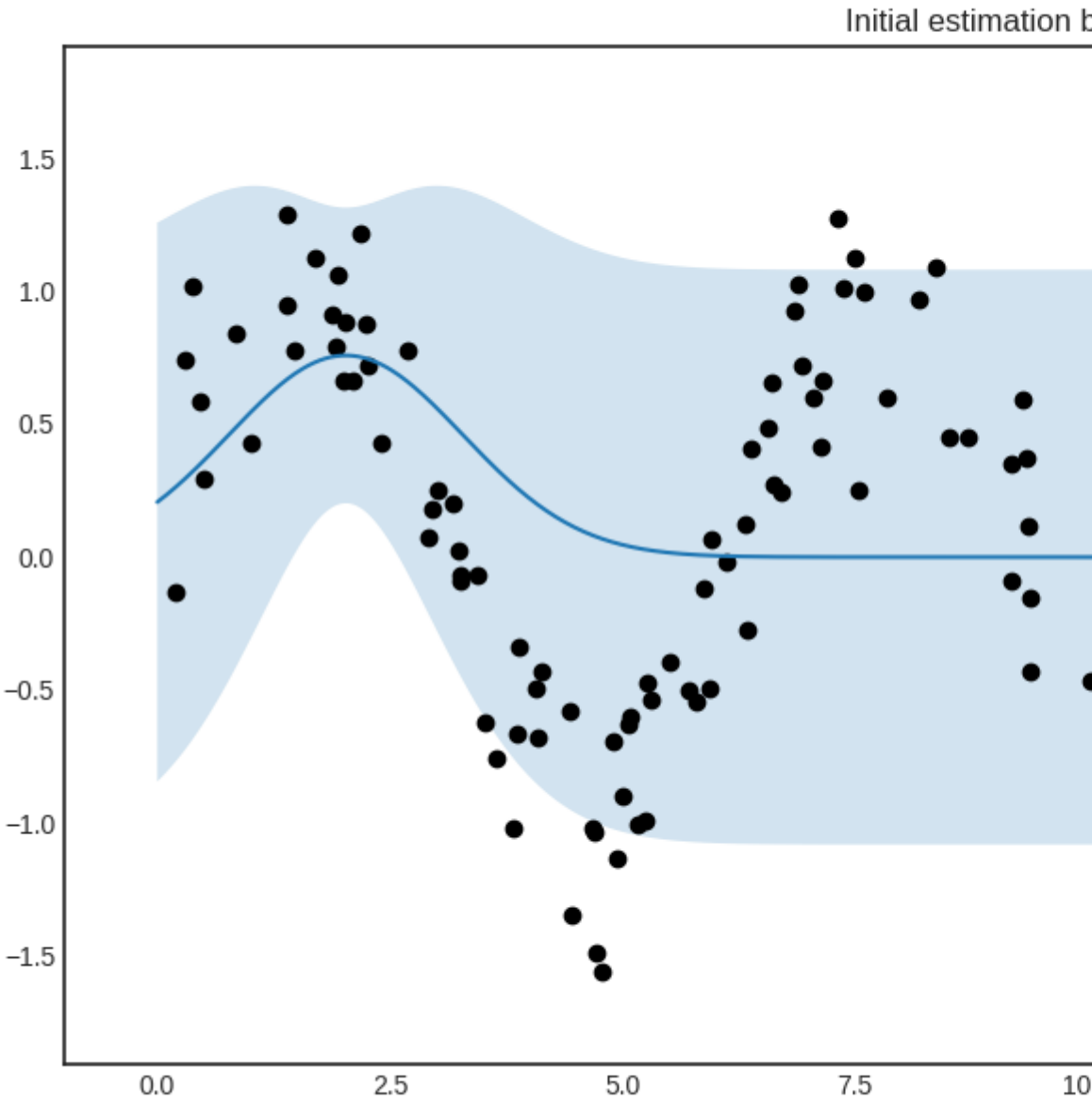
For active learning, we shall define a custom query strategy tailored to Gaussian processes. In a nutshell, a *query stategy* in modAL is a function taking (at least) two arguments (an estimator object and a pool of examples), outputting the index of the queried instance and the instance itself. In our case, the arguments are `regressor` and `X`.

```
def GP_regression_std(regressor, X):
    _, std = regressor.predict(X, return_std=True)
    query_idx = np.argmax(std)
    return query_idx, X[query_idx]
```

After setting up the query strategy and the data, the active learner can be initialized.

```
from modAL.models import ActiveLearner
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import WhiteKernel, RBF

n_initial = 5
initial_idx = np.random.choice(range(len(X)), size=n_initial, replace=False)
X_training, y_training = X[initial_idx], y[initial_idx]

kernel = RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e3)) \
         + WhiteKernel(noise_level=1, noise_level_bounds=(1e-10, 1e+1))

regressor = ActiveLearner(
    estimator=GaussianProcessRegressor(kernel=kernel),
    query_strategy=GP_regression_std,
    X_training=X_training.reshape(-1, 1), y_training=y_training.reshape(-1, 1)
)
```

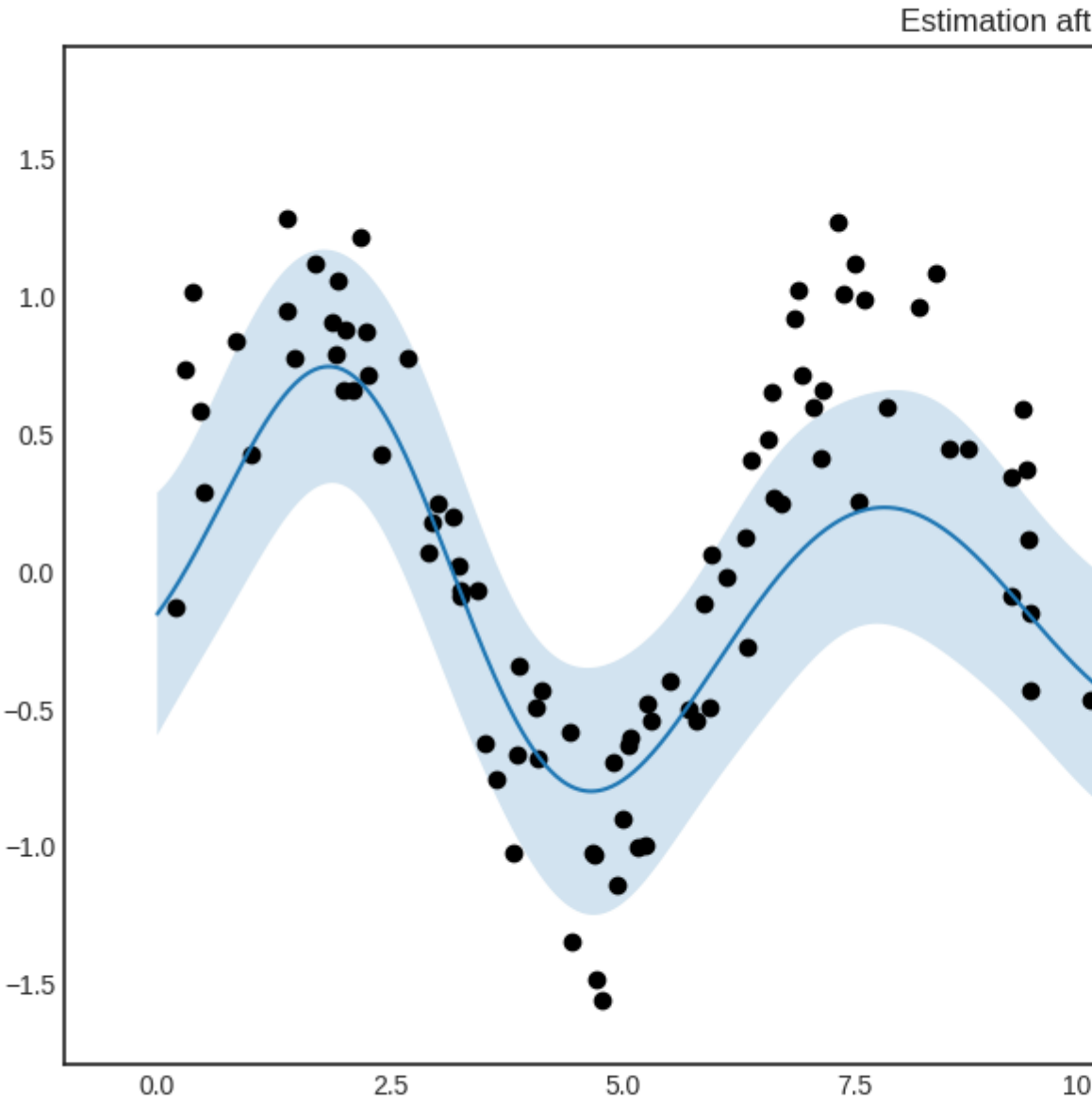
The initial regressor is not very accurate.



The blue band enveloping the regressor represents the standard deviation of the Gaussian process at the given point. Now we are ready to do active learning!

```
# active learning
n_queries = 10
for idx in range(n_queries):
    query_idx, query_instance = regressor.query(X)
    regressor.teach(X[query_idx].reshape(1, -1), y[query_idx].reshape(1, -1))
```

After a few queries, we can see that the prediction is much improved.



1.4 Citing

If you use modAL in your projects, you can cite it as

```
@article{modAL2018,
    title={mod{AL}: {A} modular active learning framework for {P}ython},
    author={Tivadar Danka and Peter Horvath},
```

(continues on next page)

(continued from previous page)

```
    url={https://github.com/cosmic-cortex/modAL},
    note={available on arXiv at \url{https://arxiv.org/abs/1805.00979}}
}
```

1.5 About the developer

modAL is developed by me, Tivadar Danka (aka cosmic-cortex in GitHub). I have a PhD in pure mathematics, but I fell in love with biology and machine learning right after I finished my PhD. I have changed fields and now I work in the Bioimage Analysis and Machine Learning Group of Peter Horvath, where I am working to develop active learning strategies for intelligent sample analysis in biology. During my work I realized that in Python, creating and prototyping active learning workflows can be made really easy and fast with scikit-learn, so I ended up developing a general framework for this. The result is modAL :) If you have any questions, requests or suggestions, you can contact me at 85a5187a@opayq.com! I hope you'll find modAL useful!

CHAPTER 2

Installation

modAL requires

- Python >= 3.5
- NumPy >= 1.13
- SciPy >= 0.18
- scikit-learn >= 0.18

You can install modAL directly with pip:

```
pip install modAL
```

Alternatively, you can install modAL directly from source:

```
pip install git+https://github.com/modAL-python/modAL.git
```

For running the examples, Matplotlib >= 2.0 is recommended.

CHAPTER 3

Extending modAL

modAL was designed for researchers, allowing quick and efficient prototyping. For this purpose, modAL makes it easy for you to use your customly designed parts, for instance query strategies or new classifier algorithms.

3.1 Building blocks of query strategies

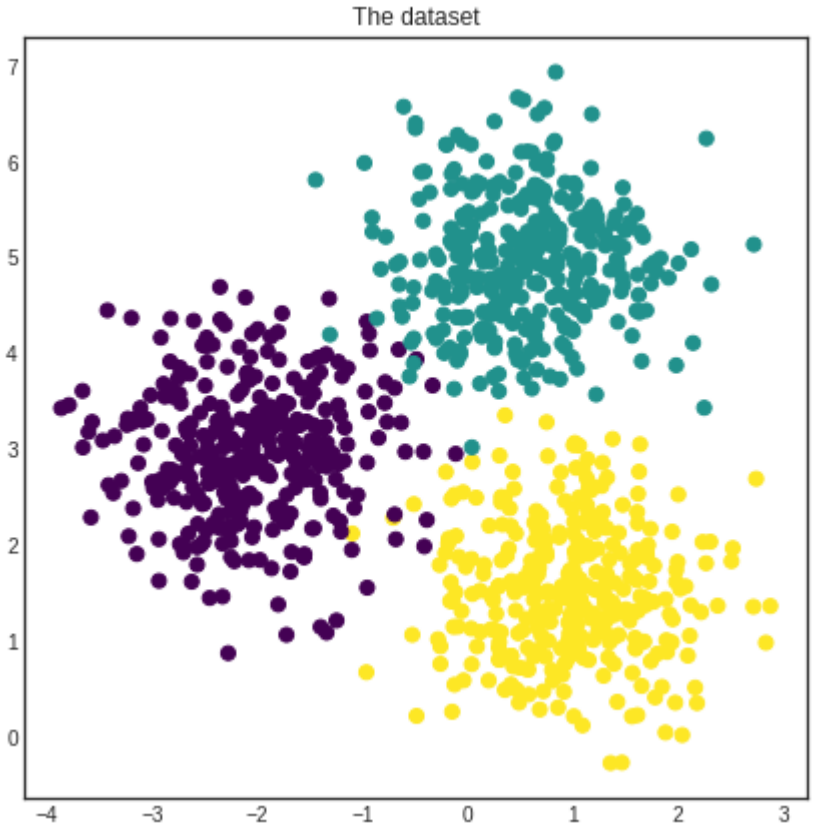
In modAL, a query strategy for active learning is implemented as a function, taking an estimator and a bunch of data, turning it into an instance from the data you supplied to it. To build elaborate custom query strategies, many building blocks are available. The two main components of a query strategy are the utility measure and the query selector. From an abstract viewpoint, this is how a query strategy looks like.

[1]:
```
def custom_query_strategy(classifier, X, a_keyword_argument=42):
    # measure the utility of each instance in the pool
    utility = utility_measure(classifier, X)

    # select the indices of the instances to be queried
    query_idx = select_instances(utility)

    # return the indices and the instances
    return query_idx, X[query_idx]
```

To demonstrate how can you create new query strategies, first we create a toy dataset.

[2]:
```
import numpy as np
from sklearn.datasets import make_blobs

# generating the data
centers = np.asarray([[-2, 3], [0.5, 5], [1, 1.5]])
X, y = make_blobs(
    n_features=2, n_samples=1000, random_state=0, cluster_std=0.7,
    centers=centers
)
```

[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# visualizing the dataset
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    plt.scatter(x=X[:, 0], y=X[:, 1], c=y, cmap='viridis', s=50)
    plt.title('The dataset')
    plt.show()
```



[4]:
```
# initial training data
initial_idx = np.random.choice(range(len(X)), size=20)
X_training, y_training = X[initial_idx], y[initial_idx]
```

Finally, we initialize an ActiveLearner instance which we will use to demonstrate our custom query strategies.

[5]:
```
from modAL.models import ActiveLearner
from sklearn.gaussian_process import GaussianProcessClassifier
from sklearn.gaussian_process.kernels import RBF

# initializing the learner
learner = ActiveLearner(
    estimator=GaussianProcessClassifier(1.0 * RBF(1.0)),
    X_training=X_training, y_training=y_training
)
```

3.2 Utility measures

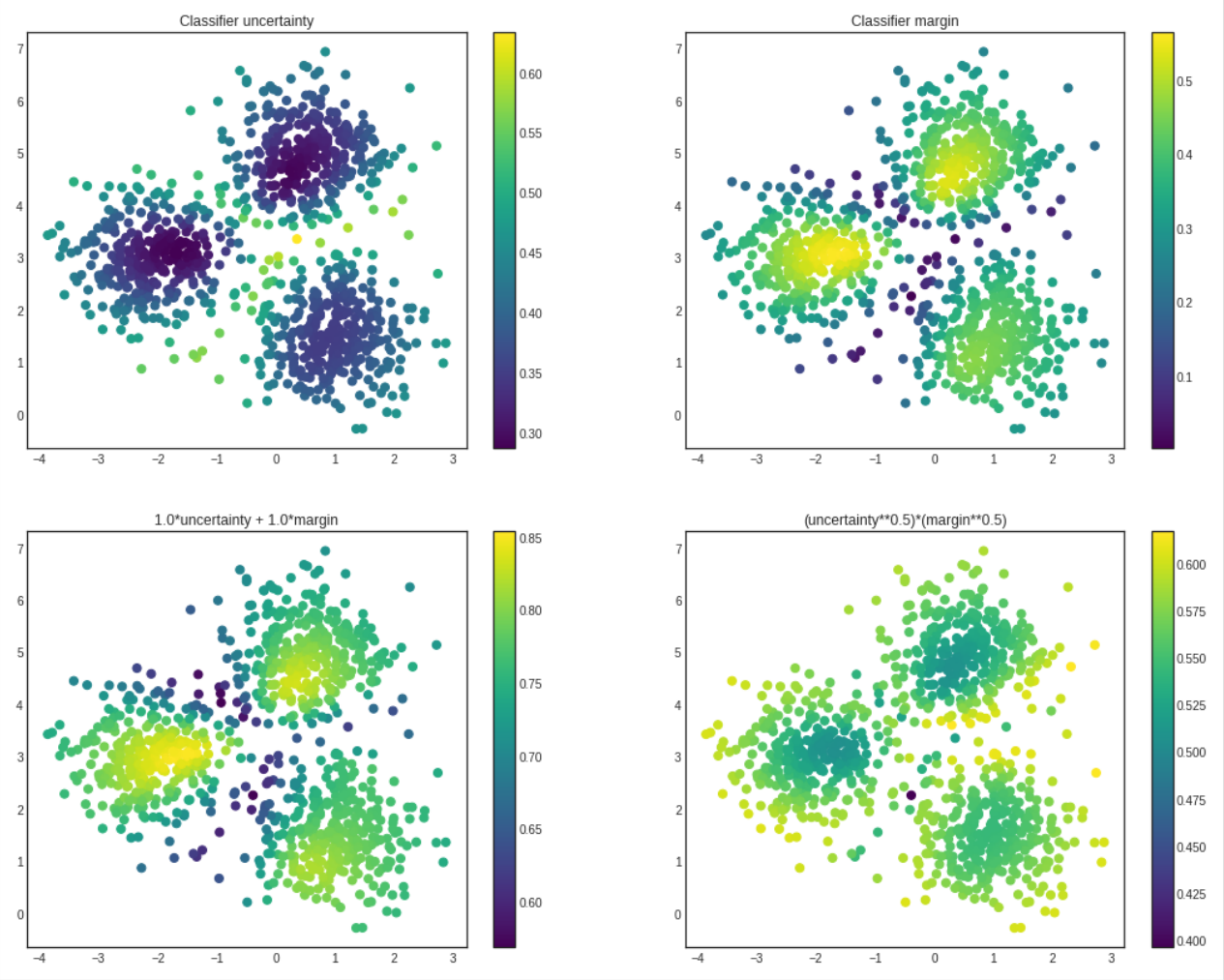
The soul of a query strategy is the utility measure. A utility measure takes a pool of examples (and frequently but optionally an estimator object) and returns a one dimensional array containing the utility score for each example. For instance, `classifier_uncertainty`, `classifier_margin` and `classifier_entropy` from `modAL.uncertainty` are utility measures which you can use. You can also implement your own or you can take linear combinations and products, as we shall see next.

3.3 Linear combinations and products

One way of creating new utility measures is to take linear combinations or products of already existing ones. For this, the function factories in the module `modAL.utils.combination` are there to help!

[6]:
```
from modAL.utils.combination import make_linear_combination, make_product
from modAL.uncertainty import classifier_uncertainty, classifier_margin

# creating new utility measures by linear combination and product
# linear_combination will return 1.0*classifier_uncertainty + 1.0*classifier_margin
linear_combination = make_linear_combination(
    classifier_uncertainty, classifier_margin,
    weights=[1.0, 1.0]
)
# product will return (classifier_uncertainty**0.5)*(classifier_margin**0.1)
product = make_product(
    classifier_uncertainty, classifier_margin,
    exponents=[0.5, 0.1]
)
```

[7]:
```
# visualizing the different utility metrics
with plt.style.context('seaborn-white'):
    utilities = [
        (1, classifier_uncertainty(learner, X), 'Classifier uncertainty'),
        (2, classifier_margin(learner, X), 'Classifier margin'),
        (3, linear_combination(learner, X), '1.0*uncertainty + 1.0*margin'),
        (4, product(learner, X), '(uncertainty**0.5)*(margin**0.1)')
    ]

    plt.figure(figsize=(18, 14))
    for idx, utility, title in utilities:
        plt.subplot(2, 2, idx)
        plt.scatter(x=X[:, 0], y=X[:, 1], c=utility, cmap='viridis', s=50)
        plt.title(title)
        plt.colorbar()

    plt.show()
```



3.4 Selectors

After the individual utility scores are calculated, a query strategy must determine which instances are to be queried. Two prebuilt selectors are available in `modAL.utils.selection`: `multi_argmax(values, n_instances=1)` selects the `n_instances` highest utility score, while `weighted_random(weights, n_instances=1)` selects the instances by random, using the supplied weighting.

3.5 Putting them together

When the components are given, putting the query strategy together is really simple.

[8]:
```
from modAL.utils.selection import multi_argmax

# defining the custom query strategy, which uses the linear combination of
# classifier uncertainty and classifier margin
def custom_query_strategy(classifier, X, n_instances=1):
    utility = linear_combination(classifier, X)
```

(continues on next page)

(continued from previous page)

```
    query_idx = multi_argmax(utility, n_instances=n_instances)
    return query_idx, X[query_idx]

custom_query_learner = ActiveLearner(
    estimator=GaussianProcessClassifier(1.0 * RBF(1.0)),
    query_strategy=custom_query_strategy,
    X_training=X_training, y_training=y_training
)
```

[9]:
```
# pool-based sampling
n_queries = 20
for idx in range(n_queries):
    query_idx, query_instance = custom_query_learner.query(X, n_instances=2)
    custom_query_learner.teach(
        X=X[query_idx].reshape(-1, 2),
        y=y[query_idx].reshape(-1, )
    )
```

This can be used immediately in the active learning workflow!

3.6 Using your custom estimators

As long as your classifier follows the scikit-learn API, you can use it in your modAL workflow. (Really, all it needs is a `.fit(X, y)` and a `.predict(X)` method.) For instance, the ensemble model implemented in Committee can be given to an ActiveLearner.

[10]:
```
from modAL.models import Committee

# initializing the learners
n_learners = 3
learner_list = []
for _ in range(n_learners):
    learner = ActiveLearner(
        estimator=GaussianProcessClassifier(1.0 * RBF(1.0)),
        X_training=X_training, y_training=y_training,
        bootstrap_init=True
    )
    learner_list.append(learner)

# assembling the Committee
committee = Committee(learner_list)

# ensemble active learner from the Committee
ensemble_learner = ActiveLearner(
    estimator=committee
)
```

CHAPTER 4

Contributing

Contributions to modAL are very much welcome! If you would like to help in general, visit the Issues page, where you'll find bugs to be fixed, features to be implemented. If you have a concrete feature in mind, you can proceed as follows.

1. Open a new issue. This helps us to discuss your idea and makes sure that you are not working in parallel with other contributors.
2. Fork the modAL repository and clone your fork to your local machine:

```
$ git clone git@github.com:username/modAL.git
```

3. Create a feature branch for the changes from the dev branch:

```
$ git checkout -b new-feature dev
```

Make sure that you create your branch from `dev`.

4. After you have finished implementing the feature, make sure that all the tests pass. The tests can be run as

```
$ python3 path-to-modAL-repo/tests/core_tests.py
```

5. Commit and push the changes.

```
$ git add modified_files
$ git commit -m 'commit message explaning the changes briefly'
$ git push origin new-feature3
```

6. Create a pull request from your fork **to the dev branch**. After the code is reviewed and possible issues are cleared, the pull request is merged to `dev`.

CHAPTER 5

ActiveLearner

In modAL, the base active learning model is the ActiveLearner class. In this short tutorial, we are going to see how to use it and what are its basic functionalities.

5.1 Initialization

To create an ActiveLearner object, you need to provide two things: a *scikit-learn estimator object* and a *query strategy function* (The latter one is optional, the default strategy is maximum uncertainty sampling.). Regarding the query strategies, you can find built-ins in `modAL.uncertainty`, but you can also implement your own. For instance, you can just simply do the following.

```
from modAL.models import ActiveLearner
from modAL.uncertainty import uncertainty_sampling
from sklearn.ensemble import RandomForestClassifier

learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    query_strategy=uncertainty_sampling
)
```

If you have initial training data available, you can train the estimator by passing it via the arguments `X_training` and `y_training`. For instance, if the samples are contained in `X_training` and the labels are in `y_training`, you can do the following.

```
learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    query_strategy=uncertainty_sampling
    X_training=X_training, y_training=y_training
)
```

After initialization, your ActiveLearner is ready to ask and learn! The learner keeps track of the training data it has seen during its lifetime.

5.2 Training

To teach newly acquired labels for the ActiveLearner, you should use the `.teach(X, y)` method. This augments the available training data with the new samples `X` and new labels `y`, then refits the estimator to this augmented training dataset. Just like this:

```
learner.teach(X, y)
```

If you would like to start from scratch, you can use the `.fit(X, y)` method to make the learner forget everything it has seen and fit the model to the newly provided data.

To train only on the newly acquired data, you should pass `only_new=True` to the `.teach()` method. This is useful when the `.fit()` method of the classifier does not retrain the model from scratch, like in Keras.

5.3 Bootstrapping

Training is also available with bootstrapping by passing `bootstrap=True` for `learner.teach()` or `learner.fit()`. In this case, a random set is sampled with replacement from the training data available (or the data provided in the case of `.fit()`), which is used to train the estimator. Bootstrapping is mostly useful when building ensemble models with bagging, for instance in a *query by committee* setting.

5.4 Querying for labels

Active learners are called *active* because if you provide them unlabelled samples, they can select you the best instances to label. In modAL, you can achieve this by calling the `.query(X)` method:

```
query_idx, query_sample = learner.query(X)

# ...obtaining new labels from the Oracle...

learner.teach(query_sample, query_label)
```

Under the hood, the `.query(X)` method simply calls the *query strategy function* specified by you upon initialization of the ActiveLearner.

The available built-in query strategies are *max uncertainty sampling*, *max margin sampling* and *entropy sampling*. For more details, see the Uncertainty-sampling page.

5.5 Query strategies

In modAL, currently there are three built-in query strategies: *max uncertainty*, *max margin* and *max entropy*, they are located in the `modAL.uncertainty` module. You can find an informal tutorial about them at the page Uncertainty-sampling.

5.6 Prediction and scoring

To use the ActiveLearner for prediction and to calculate the mean accuracy score, you can just do what you would do with a *scikit-learn* classifier: call the `.predict(X)` and `.score(X, y)` methods. If you would like to use more

sophisticated metrics for your prediction, feel free to use a function from `sklearn.metrics`, they are compatible with modAL.

CHAPTER 6

BayesianOptimizer

When a function is expensive to evaluate, or when gradients are not available, optimalizing it requires more sophisticated methods than gradient descent. One such method is Bayesian optimization, which lies close to active learning. In Bayesian optimization, instead of picking queries by maximizing the uncertainty of predictions, function values are evaluated at points where the promise of finding a better value is large. In modAL, these algorithms are implemented with the `BayesianOptimizer` class, which is a sibling of `ActiveLearner`. They are both children of the `BaseLearner` class and they have the same interface, although their uses differ. In the following, we are going to shortly review this.

6.1 Differences with ActiveLearner

Initializing a `BayesianOptimizer` is syntactically identical to the initialization of `ActiveLearner`, although there are a few important differences.

```
from modAL.models import BayesianOptimizer
from modAL.acquisition import max_EI
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern

kernel = Matern(length_scale=1.0)
regressor = GaussianProcessRegressor(kernel=kernel)

optimizer = BayesianOptimizer(
    estimator=regressor,
    query_strategy=max_EI
)
```

Most importantly, `BayesianOptimizer` works with a regressor. You can use them with a classifier if the labels are numbers, but the result will be meaningless. Bayesian optimization typically uses a Gaussian process regressor to keep a hypothesis about the function to be optimized and estimate the expected gains when a certain point is picked for evaluation. This latter is the task of the acquisition function. (*See below for details.*)

The actual optimization loop is identical to the one you would use with the `ActiveLearner`.

```
# Bayesian optimization: func is to be optimized
for n_query in range(n_queries):
    query_idx, query_inst = optimizer.query(X)
    optimizer.teach(X[query_idx].reshape(1, -1), func(X[query_idx]).reshape(1, -1))
```

Again, the bottleneck in Bayesian learning is not necessarily the availability of labels. The function to be optimized can take a long time and a lot of money to evaluate. For instance, when optimizing the hyperparameters of a deep neural network, evaluating the accuracy of the model can take a few days of training. This is a case when Bayesian optimization is very useful. For more details, see this paper for instance.

To see the maximum value so far, use `optimizer.get_max()`:

```
X_max, y_max = optmizer.get_max()
```

6.2 Acquisition functions

Currently, there are three built in acquisition functions in the `modAL.acquisition` module: *expected improvement*, *probability of improvement* and *upper confidence bounds*. You can find more information about them at the page *Acquisition functions*.

CHAPTER 7

Committee

One of the popular active learning strategies is the *Query by Committee*, where we keep several hypotheses (i.e. trained classifiers) about the data, and we select our queries by measuring the disagreement of the hypotheses. In modAL, this model is implemented in the Committee class.

7.1 Initialization

To create a Committee object, you need to provide two things: a list of *ActiveLearner* objects and a *query strategy function*. (A list of *scikit-learn estimators* won't suffice, because each learner needs to keep track of the training examples it has seen.) For instance, you can do the following.

```
from modAL.models import Committee
from modAL.disagreement import vote_entropy_sampling

# a list of ActiveLearners:
learners = [learner_1, learner_2]

committee = Committee(
    learner_list=learners,
    query_strategy=vote_entropy_sampling
)
```

You can find the disagreement-based query strategies in `modAL.disagreement`, although a Committee works with uncertainty sampling strategies also. (Using a Committee with uncertainty sampling can be useful, for instance if you would like to build an active regression model and you need an ensemble of regressors to estimate uncertainty.)

7.2 Iterating through the learners of the Committee

Since Committee is iterable, feel free to use a `for` loop just as you would in any other case:

```
for learner in committee:
    # ...do something with the learner...
```

`len(committee)` also works, it returns the number of learners in the Committee.

7.3 Training and querying for labels

Training and querying for labels work exactly the same as for the ActiveLearner. To teach new examples for the Committee, you should use the `.teach(X, y)` method, where `X` contains the new training examples and `y` contains the corresponding labels or values. This teaches the new training example for all learners in the Committee, hopefully improving performance.

To select the best instances to label, use the `.query(X)` method, just like for the ActiveLearner. It simply calls the query strategy function you specified, which measures the utility for each sample and selects the ones having the highest utility.

7.4 Bagging

When building ensemble models such as in the Query by Committee setting, bagging can be useful and can improve performance. In Committee, this can be done with the methods `.bag(X, y)` and `.rebag()`. The difference between them is that `.bag(X, y)` makes each learner forget the data it has seen until this point and replaces it with `X` and `y`, while `.rebag()` refits each learner in the Committee by bootstrapping its training instances but leaving them as they were.

Bagging is also available during teaching new examples by passing `bootstrap=True` for the `.teach()` method. Just like this:

```
committee.teach(X_new, y_new, bootstrap=True)
```

First, this stores the new training examples and labels in each learner, then fits them using a bootstrapped subset of the known examples for the learner.

7.5 Query strategies

Currently, there are three built-in query by committee strategies in modAL: *max vote entropy*, *max uncertainty entropy* and *max disagreement*. They are located in the `modAL.disagreement` module. You can find an informal tutorial at the page *Disagreement sampling*.

7.6 Voting and predicting

Although the API for predicting is the same for Committee than the ActiveLearner class, they are working slightly differently under the hood.

To obtain the predictions and class probabilities *for each learner* in the Committee, you should use the `.vote(X)` and `.vote_proba(X)` methods, where `X` contains the samples to be predicted.

`.vote(X)` returns a numpy array of shape [n_samples, n_learners] containing the class predictions according to each learner, while `.vote_proba(X)` returns a numpy array of shape [n_samples, n_learners, n_classes] containing the class probabilities for each learner. You don't need to worry about different learners seeing a different set of class labels, Committee is smart about that.

To get the predictions and class probabilities of the Committee itself, you shall use the `.predict(X)` and `.predict_proba()` methods, they are used like their `scikit-learn` relatives. `.predict_proba()` returns the class probability averaged across each learner (the so-called *consensus probabilities*), while `.predict()` selects the most likely label based on the consensus probabilities.

CHAPTER 8

CommitteeRegressor

In addition to committee based active learning for classification, modAL also implements committee based regression. This is done with the CommitteeRegressor class.

8.1 Differences between Committee and CommitteeRegressor

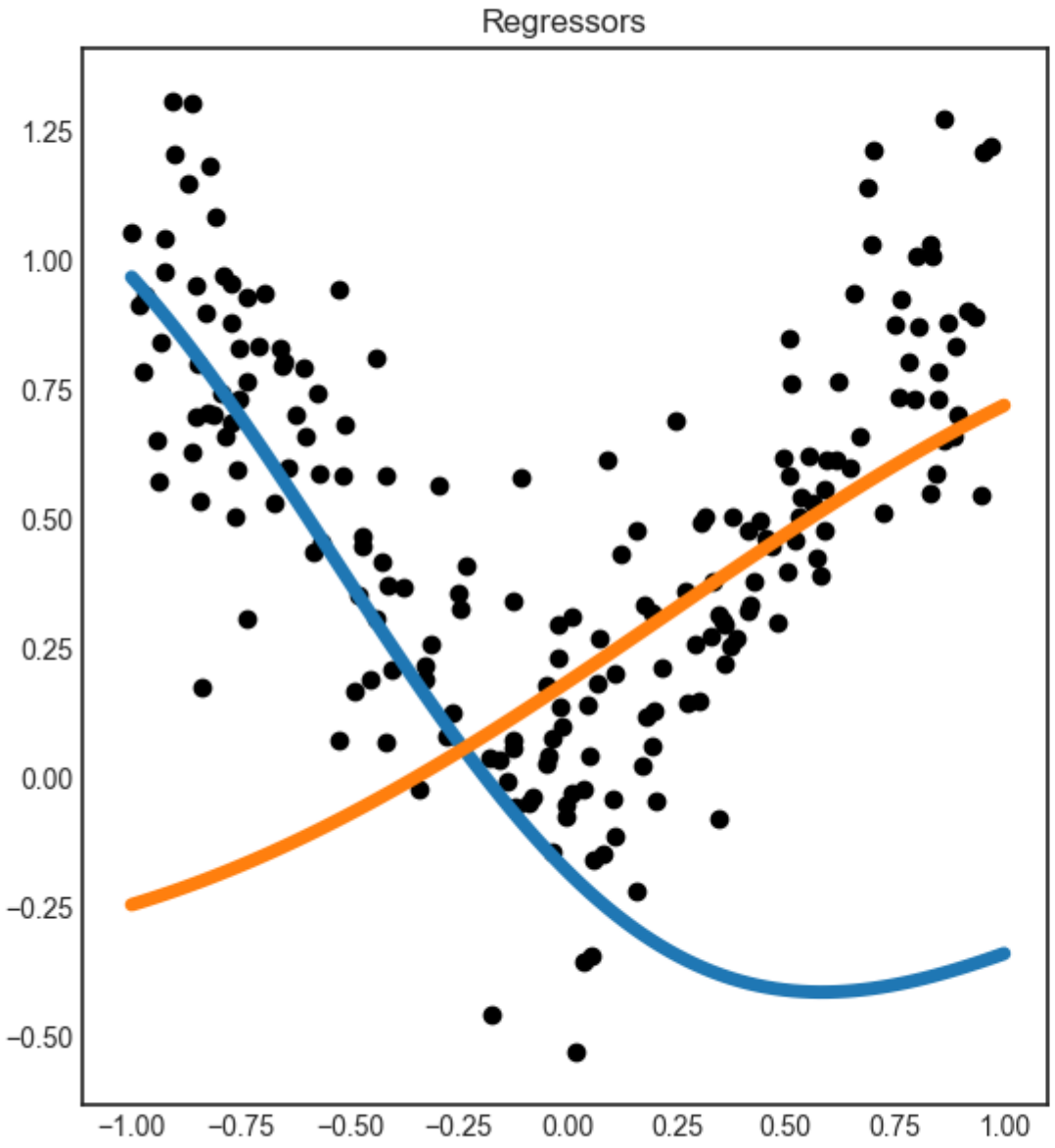
From an API viewpoint, there are two main differences between the CommitteeRegressor and the Committee classes. First, CommitteeRegressor doesn't have `.vote_proba()` and `.predict_proba()` methods, since regressors in general doesn't provide a way to estimate the probability of correctness. (One notable exception is the Gaussian process regressor.)

The other main difference is that now you can pass the argument `return_std=True` for the method `.predict()`, which in this case will return the standard deviation of the prediction. This follows the scikit-learn API for those regressor objects which the standard deviation can be calculated.

8.2 Measuring disagreement

With an ensemble of regressors like in the CommitteeRegressor model, a measure of disagreement can be the standard deviation of predictions, which provides a simple way to query for labels. This is not the case in general: for ordinary regressors, it is difficult to come up with viable query strategies because they don't always provide a way to measure uncertainty. (One notable exception is the Gaussian process regressor.)

This is demonstrated in this example, where two regressors are trained on distinct subsets of the same dataset. In the figure below, the regressors are shown along with the mean predictions and the standard deviation.


Regressors
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

CHAPTER 9

Acquisition functions

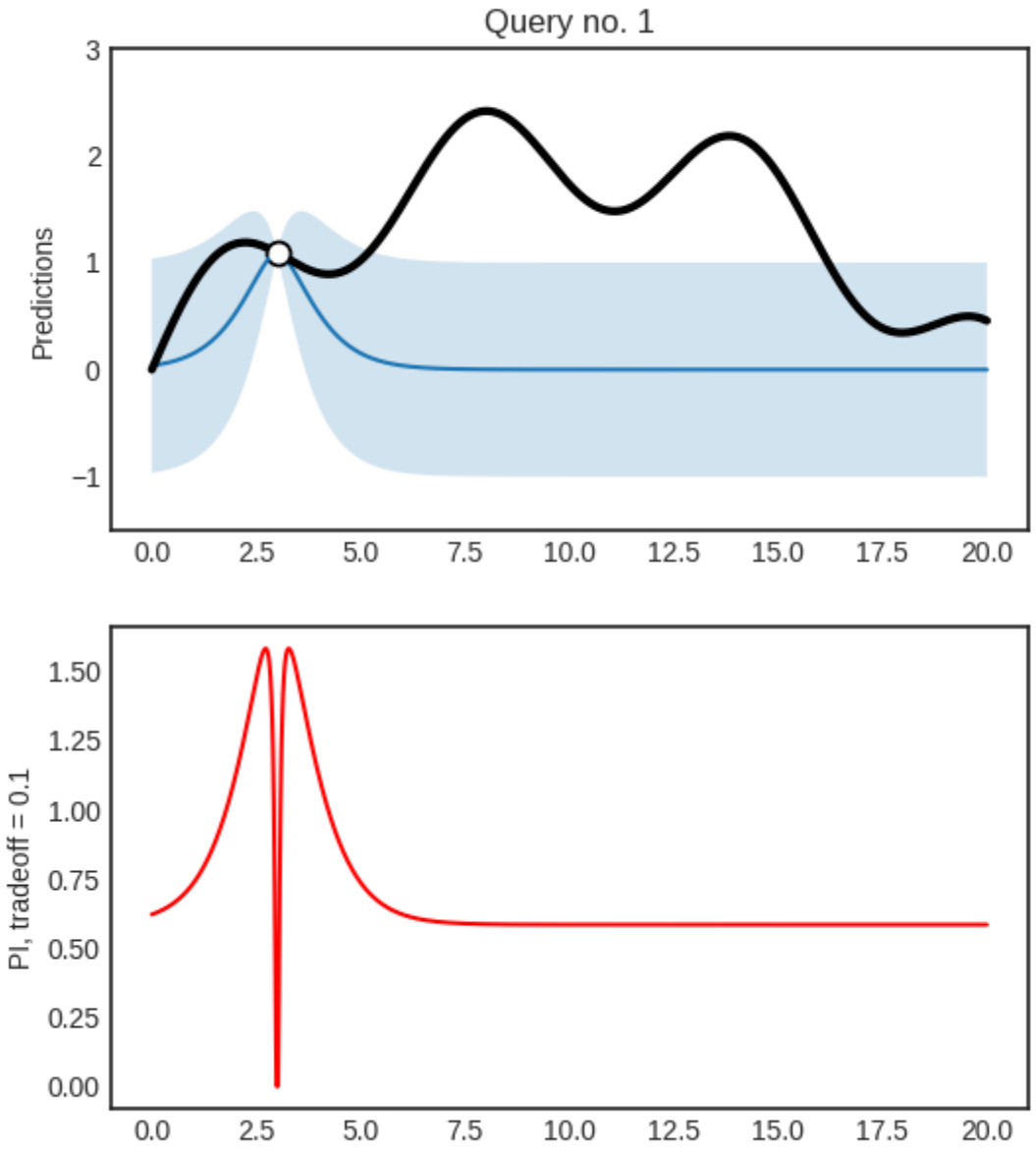
In Bayesian optimization, a so-called *acquisition funciton* is used instead of the uncertainty based utility measures of active learning. In modAL, Bayesian optimization algorithms are implemented in the `modAL.models.BayesianOptimizer` class. Currently, there are three available acquisition functions: probability of improvement, expected improvement and upper confidence bound.

9.1 Probability of improvement

The probability of improvement is defined by

$$PI(x) = \psi\Big(\frac{\mu(x) - f(x^+) - \xi}{\sigma(x)}\Big),$$

where $\mu(x)$ and $\sigma(x)$ are the mean and variance of the regressor at x, f is the function to be optimized with estimated maximum at x^+, ξ is a parameter controlling the degree of exploration and $\psi(z)$ denotes the cumulative distribution function of a standard Gaussian distribution.



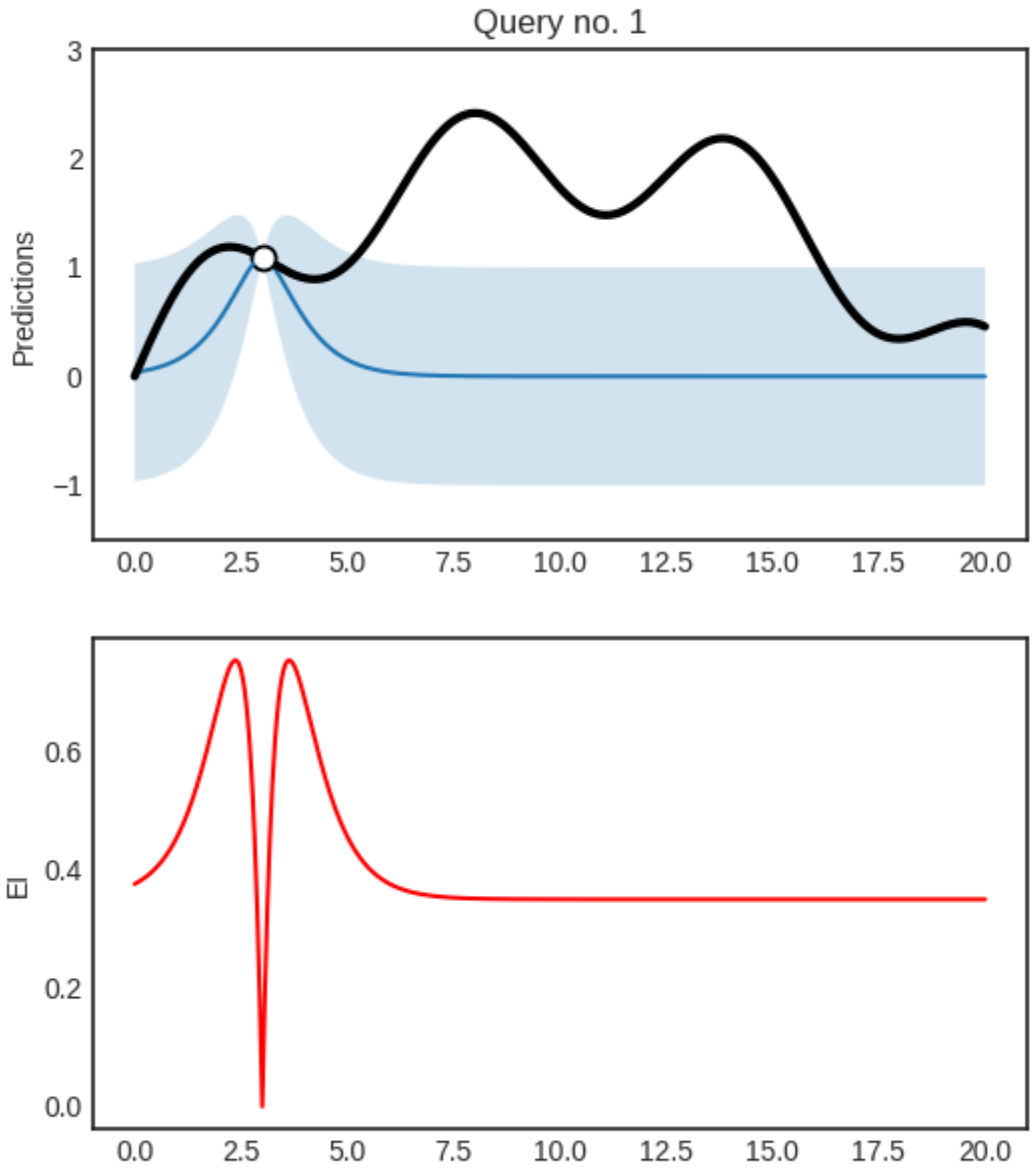
If you would like to use it with a BayesianOptimizer‘‘, you should pass `modAL.acquisition.max_PI` as the query strategy upon initialization.

9.2 Expected improvement

The expected improvement is defined by

$$\begin{aligned} EI(x) =& (\mu(x) - f(x^+) - \xi)\psi\Big(\frac{\mu(x) - f(x^+) - \xi}{\sigma(x)}\Big) \\ &+ \sigma(x)\phi\Big(\frac{\mu(x) - f(x^+) - \xi}{\sigma(x)}\Big), \end{aligned}$$

where $\mu(x)$ and $\sigma(x)$ are the mean and variance of the regressor at x, f is the function to be optimized with estimated maximum at x^+, ξ is a parameter controlling the degree of exploration and $\psi(z)$, $\phi(z)$ denotes the cumulative distribution function and density function of a standard Gaussian distribution.



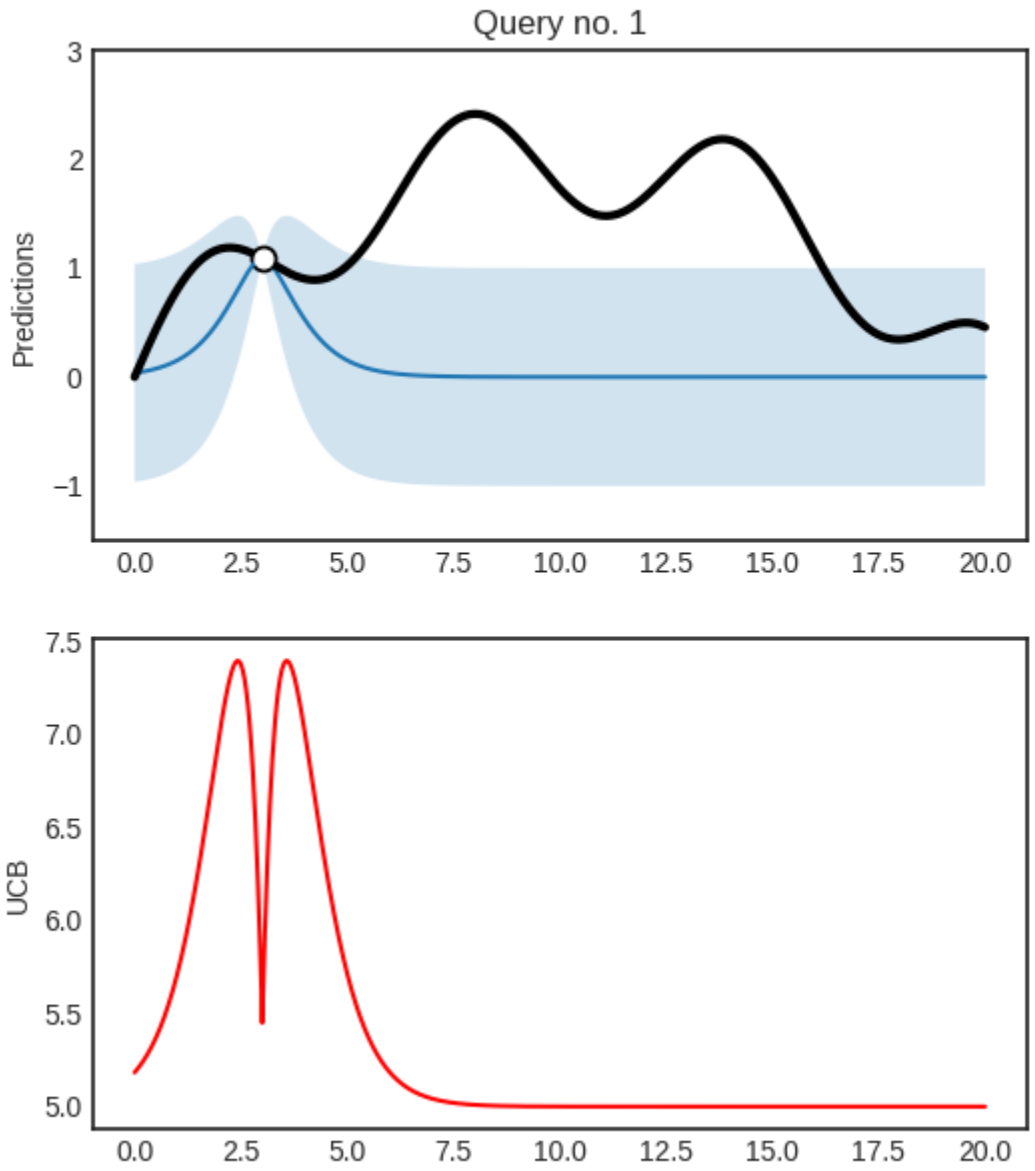
If you would like to use it with a `BayesianOptimizer`, you should pass `modAL.acquisition.max_EI` as the query strategy upon initialization.

9.3 Upper confidence bound

The upper confidence bound is defined by

$$UCB(x) = \mu(x) + \beta\sigma(x),$$

where $\mu(x)$ and $\sigma(x)$ are the mean and variance of the regressor and β is a parameter controlling the degree of exploration.



If you would like to use it with a `BayesianOptimizer`, you should pass `modAL.acquisition.max_UCB` as the query strategy upon initialization.

CHAPTER 10

Uncertainty sampling

When you present unlabelled examples to an active learner, it finds you the most *useful* example and presents it for you to be labelled. This is done by first calculating the *usefulness* of prediction (whatever it means) for each example and select an instance based on the usefulness. The thing is, there are several ways to measure this. They are based upon the classification uncertainty, hence they are called *uncertainty measures*. In modAL, currently you can select from three built-in measures: *classification uncertainty*, *classification margin* and *classification entropy*. In this quick tutorial, we are going to review them. For more details, see Section 2.3 of the awesome book Active learning by Burr Settles!

[1]:
```
import numpy as np
```

10.1 Classification uncertainty

The simplest measure is the uncertainty of classification defined by

$$U(x) = 1 - P(\hat{x}|x)$$

where x is the instance to be predicted and $\hat{x}$ is the most likely prediction.

For example, if you have classes `[0, 1, 2]` and classification probabilities `[0.1, 0.2, 0.7]`, the most likely class according to the classifier is `2` with uncertainty 0.3. If you have three instances with class probabilities

[2]:
```
proba = np.array([[0.1 , 0.85, 0.05],
                  [0.6 , 0.3 , 0.1 ],
                  [0.39, 0.61, 0.0 ]])
```

the corresponding uncertainties are:

[3]:
```
1 - proba.max(axis=1)
```

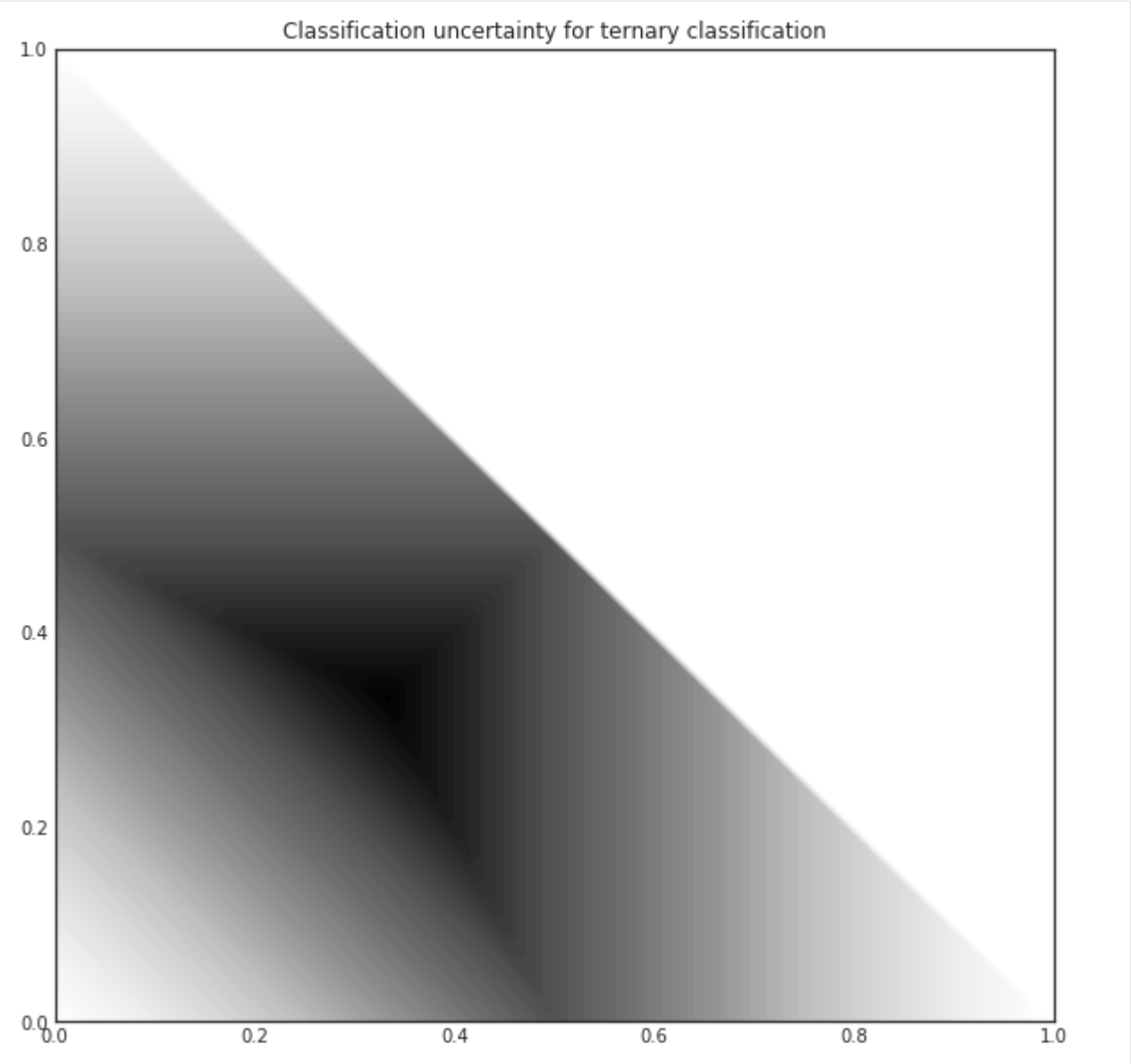
[3]:
```
array([0.15, 0.4 , 0.39])
```

In the above example, the most uncertain sample is the second one. When querying for labels based on this measure, the strategy selects the sample with the highest uncertainty.

For this ternary classification problem, given the first two probabilities, the classification uncertainty looks like the following.

[4]:
```
import matplotlib.pyplot as plt
%matplotlib inline

from itertools import product
n_res = 100
p1, p2 = np.meshgrid(np.linspace(0, 1, n_res), np.linspace(0, 1, n_res))
p3 = np.maximum(1 - p1 - p2, 0)
```

[5]:
```
uncertainty = 1 - np.maximum.reduce([p1, p2, p3])

with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 10))
    plt.contourf(p1, p2, uncertainty*((p1+p2) < 1), 100)
    plt.title('Classification uncertainty for ternary classification')
```



10.2 Classification margin

Classification margin is the difference in probability of the first and second most likely prediction, that is, it is defined by

$$M(x) = P(\hat{x_1}|x) - P(\hat{x_2}|x)$$

where $\hat{x_1}$ and $\hat{x_2}$ are the first and second most likely classes. Using the same example we used for classification uncertainty, if the class probabilities are

[6]:
```
proba = np.array([[0.1 , 0.85, 0.05],
                  [0.6 , 0.3 , 0.1 ],
                  [0.39, 0.61, 0.0 ]])
```

the corresponding margins are:

[7]:
```
part = np.partition(-proba, 1, axis=1)
margin = - part[:, 0] + part[:, 1]
```

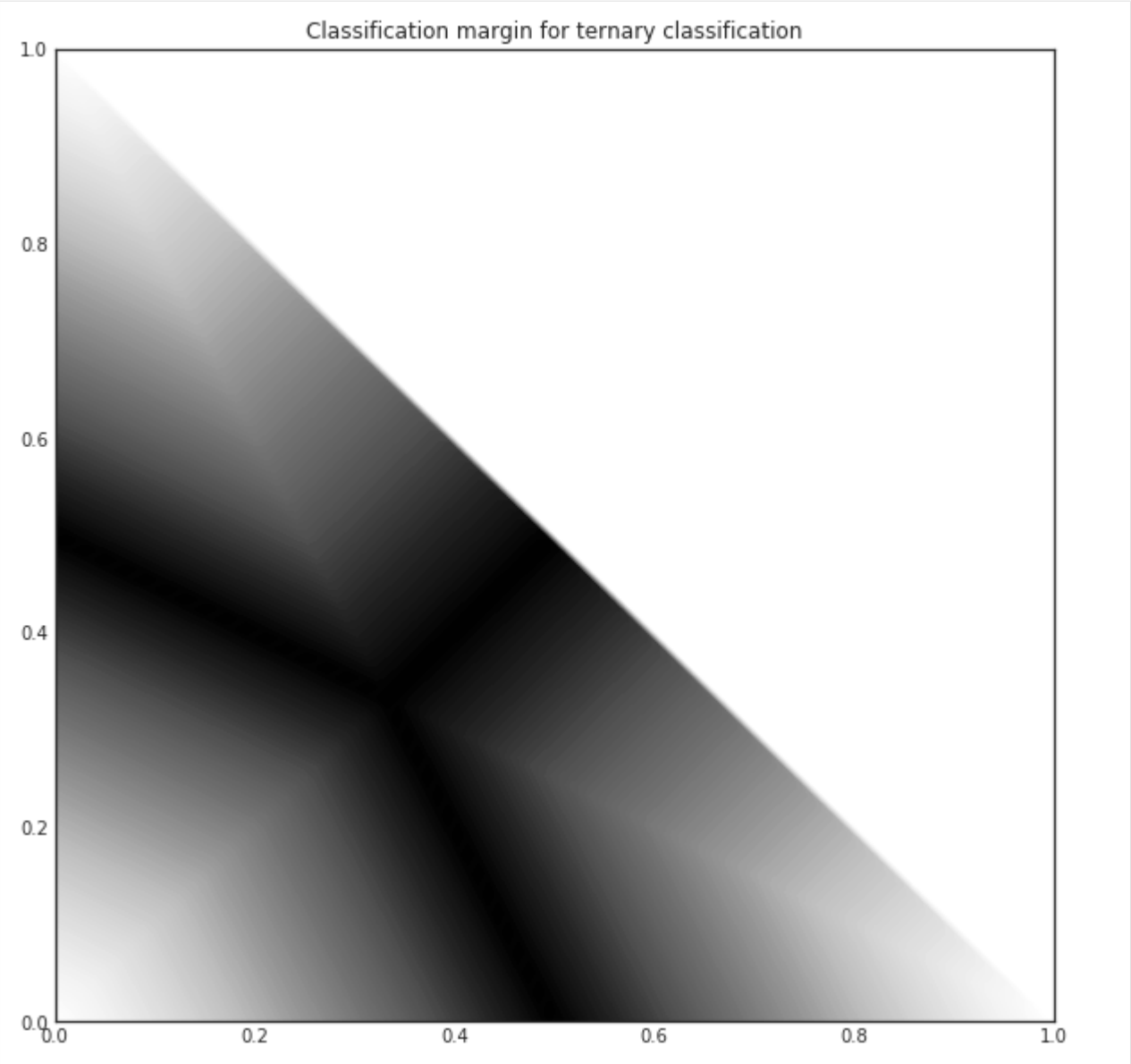
[8]:
```
margin
```

[8]:
```
array([0.75, 0.3 , 0.22])
```

When querying for labels, the strategy selects the sample with the *smallest* margin, since the smaller the decision margin is, the more unsure the decision. In this case, it would be the third sample. For this ternary classification problem, the classifier margin plotted against the first two probabilities are the following.

[9]:
```
proba = np.vstack((p1.ravel(), p2.ravel(), p3.ravel())).T

part = np.partition(-proba, 1, axis=1)
margin = - part[:, 0] + part[:, 1]

margin = margin.reshape(p1.shape)
```

[10]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 10))
    plt.contourf(p1, p2, (1-margin)*((p1+p2) < 1), 100)
    plt.title('Classification margin for ternary classification')
```



10.3 Classification entropy

The third built-in uncertainty measure is the classification entropy, which is defined by

$$H(x) = -\sum_k p_k \log(p_k)$$

where p_k is the probability of the sample belonging to the k-th class. Heuristically, the entropy is proportional to the average number of guesses one has to make to find the true class. In our usual example

[11]:
```
proba = np.array([[0.1 , 0.85, 0.05],
                  [0.6 , 0.3 , 0.1 ],
                  [0.39, 0.61, 0.0 ]])
```

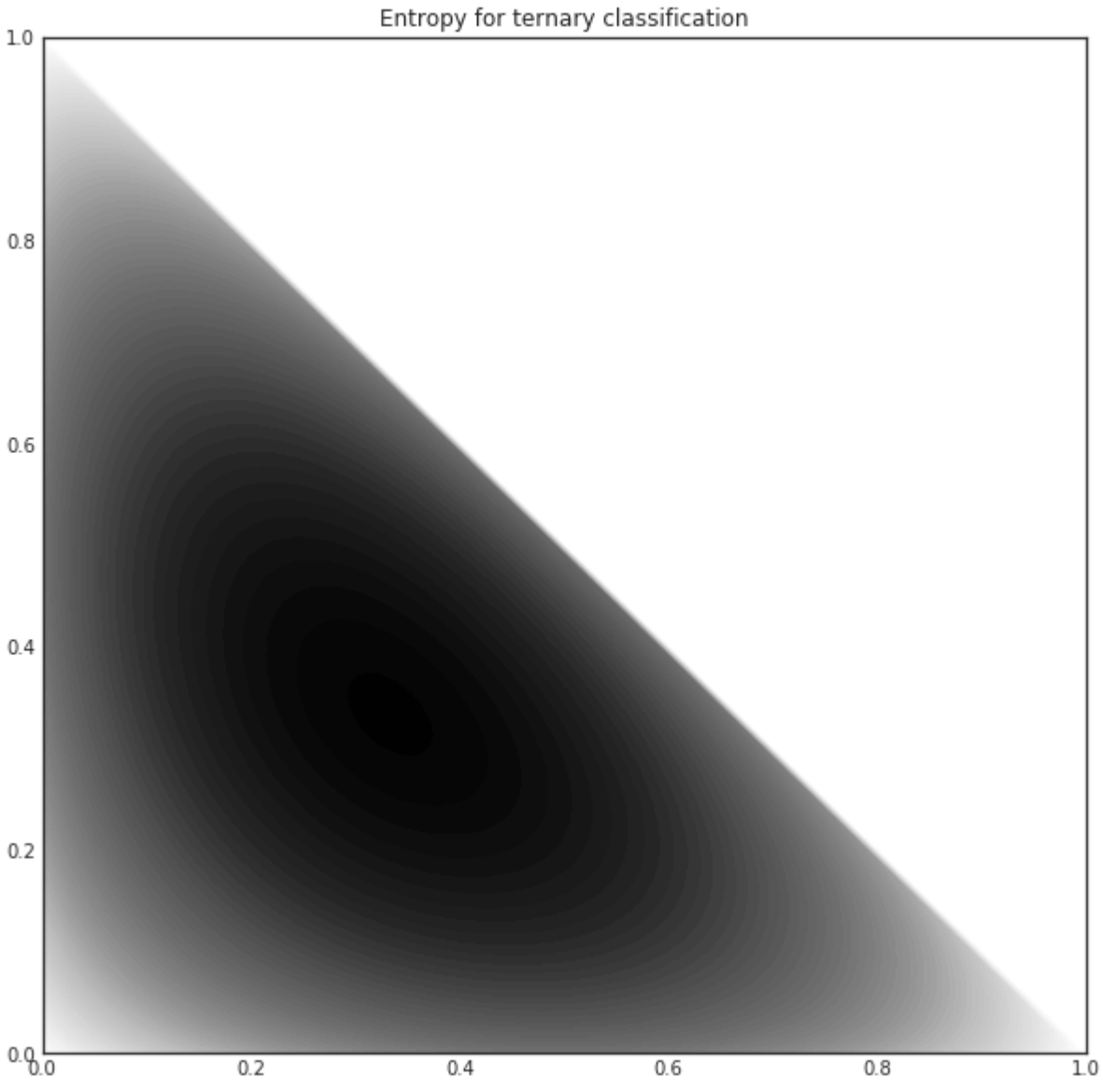
the corresponding entropies are

[12]:
```
from scipy.stats import entropy

entropy(proba.T)
```

[12]:
```
array([0.51818621, 0.89794572, 0.66874809])
```

The closer the distribution to uniform, the larger the entropy. Again, if we plot the entropy against the first two probabilities of a ternary classification problem, we obtain the following.

[13]:
```
proba = np.vstack((p1.ravel(), p2.ravel(), p3.ravel())).T
entr = entropy(proba.T).reshape(p1.shape)

with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 10))
    plt.contourf(p1, p2, entr*((p1+p2) < 1), 100)
    plt.title('Entropy for ternary classification')
```



CHAPTER 11

Disagreement sampling

When you have several hypotheses about your data, selecting the next instances to label can be done by measuring the disagreement between the hypotheses. Naturally, there are many ways to do that. In modAL, there are three built-in disagreement measures and query strategies: *vote entropy*, *consensus entropy* and *maximum disagreement*. In this quick tutorial, we are going to review them. For more details, see Section 3.4 of the awesome book Active learning by Burr Settles.

11.1 Disagreement sampling for classifiers

Committee-based models in modAL come in two flavors: Committee for classifiers and CommitteeRegressor for regressors. First, let's take a look at disagreement-based sampling strategies for classification!

11.1.1 Vote entropy

Suppose that you have a Committee of three classifiers, classes `[0, 1, 2]` and five instances to classify. If you would like to calculate the vote entropy, first you ask every classifier about its prediction:

```
>>> vote
... [[0, 1, 0]
...  [1, 1, 2]
...  [0, 1, 2]
...  [2, 2, 2]
...  [1, 2, 2]]
```

Each instance has a corresponding probability distribution to it: the distribution of class labels when picking a classifier by random. In the first instance, there are two votes for `0`, one votes for `1` and zero votes for `2`. In this case, this distribution is `[0.6666, 0.3333, 0.0]`. The distributions for all instances are

```
>>> p_vote
... [[0.6666, 0.3333, 0.0   ]
...  [0.0   , 0.6666, 0.3333]
```

(continues on next page)

(continued from previous page)

```
...  [0.3333, 0,3333, 0,3333]
...  [0.0   , 0.0   , 1.0   ]
...  [0.0   , 0.3333, 0.6666]]
```

Vote entropy selects the instance for which the entropy of this vote distribution is the largest. In this case, the vote entropies are

```
>>> vote_entropy
... [0.6365, 0.6365, 1.0986, 0.0, 0.6365]
```

Since all three votes are different for the third instance, the entropy is the largest there, thus a vote entropy based query strategy would choose that one.

11.1.2 Consensus entropy

Instead of calculating the distribution of the votes, the *consensus entropy* disagreement measure first calculates the average of the class probabilities of each classifier. This is called the consensus probability. Then the entropy of the consensus probability is calculated and the instance with the largest consensus entropy is selected.

For an example, let's suppose that we continue the previous example with three classifiers, classes `[0, 1, 2]` and five instances to classify. For each classifier, the class probabilities are

```
>>> vote_proba
... [[[0.8, 0.1, 0.0]    # \
...   [0.3, 0.7, 0.0]    # |
...   [1.0, 0.0, 0.0]    # |  <-- class probabilities for the first classifier
...   [0.2, 0.2, 0.6]    # |
...   [0.2, 0.7, 0.1]],  # /
...  [[0.0, 1.0, 0.0]    # \
...   [0.4, 0.6, 0.0]    # |
...   [0.2, 0.7, 0.1]    # |  <-- class probabilities for the second classifier
...   [0.3, 0.1, 0.6]    # |
...   [0.0, 0.0, 1.0]],  # /
...  [[0.7, 0.2, 0.1]    # \
...   [0.4, 0.0, 0.6]    # |
...   [0.3, 0.2, 0.5]    # |  <-- class probabilities for the third classifier
...   [0.1, 0.0, 0.9]    # |
...   [0.0, 0.1, 0.9]]]  # /
```

In this case, the consensus probabilities are

```
>>> consensus_proba
... [[0.5   , 0.4333, 0.0333]
...  [0.3666, 0.4333, 0.2   ]
...  [0.5   , 0.3   , 0.2   ]
...  [0.2   , 0.1   , 0.7   ]
...  [0.0666, 0.2666, 0.6666]]
```

The entropy of this is

```
>>> consensus_entropy
... [0.8167, 1.0521, 1.0296, 0.8018, 0.8033]
```

Even though the votes for the second instance are `[1, 1, 2]`, since the classifiers are quite unsure, thus the consensus entropy is high. In this case, the query strategy would select the second example to be labelled by the Oracle.

11.1.3 Max disagreement

The disagreement measures so far take the actual *disagreement* into account in a weak way. Instead of this, it is possible to measure each learner's disagreement with the consensus probabilities and query the instance where the disagreement is largest for some learner. This is called *max disagreement sampling*. Continuing our example, if the vote probabilities for each learner and the consensus probabilities are given, we can calculate the Kullback-Leibler divergence of each learner to the consensus prediction and then for each instance, select the largest value.

```
>>> for i in range(5):
>>>     for j in range(3):
>>>         learner_KL_div[i, j] = entropy(vote_proba[j, i], qk=consensus_proba[i])
>>>
>>> learner_KL_div
... [[0.32631363,  0.80234647,  0.15685227],
...  [0.27549995,  0.23005799,  0.69397192],
...  [0.69314718,  0.34053564,  0.22380466],
...  [0.04613903,  0.02914912,  0.15686827],
...  [0.70556709,  0.40546511,  0.17201121]]
>>>
>>> max_disagreement
... [0.80234647,  0.69397192,  0.69314718,  0.15686827,  0.70556709]
```

In this case, one of the learner highly disagrees with the others in the class of the first instance. Thus, the max disagreement sampling would choose this one to be labelled by the Oracle.

11.2 Disagreement sampling for regressors

Since regressors, in general, don't provide a way to calculate prediction probabilities, disagreement measures for classifiers may not work with regressors. Despite this, ensemble regression models can be always used in an active learning scenario, because the standard deviation of the predictions at a given point can be thought of as a measure of disagreement.

11.2.1 Standard deviation sampling

When a committee of regressors is available, the uncertainty of predictions can be estimated by calculating the standard deviation of predictions. This is done by the `modAL.disagreement.max_std_sampling` function.

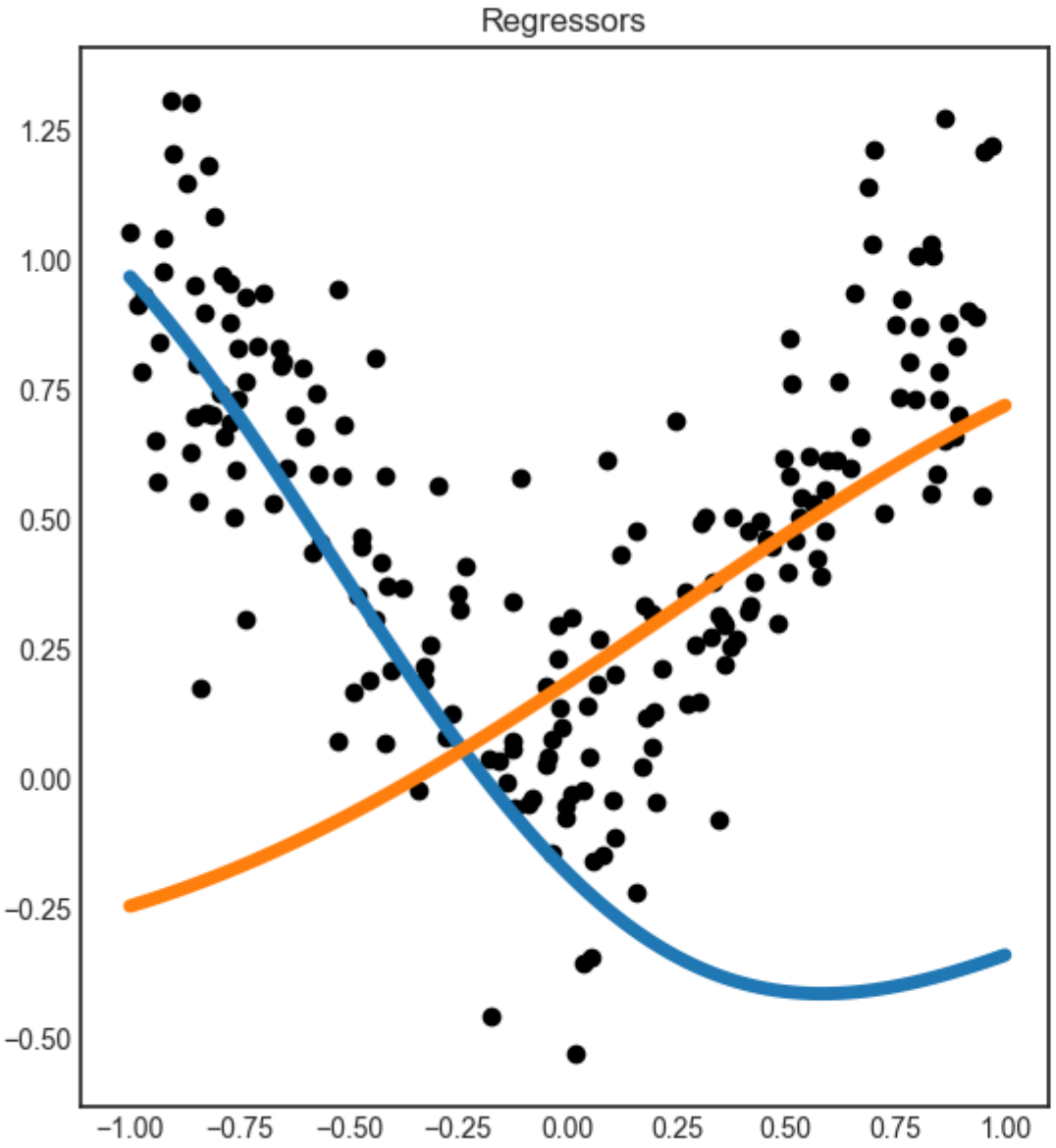
11.3 Disagreement measures in action

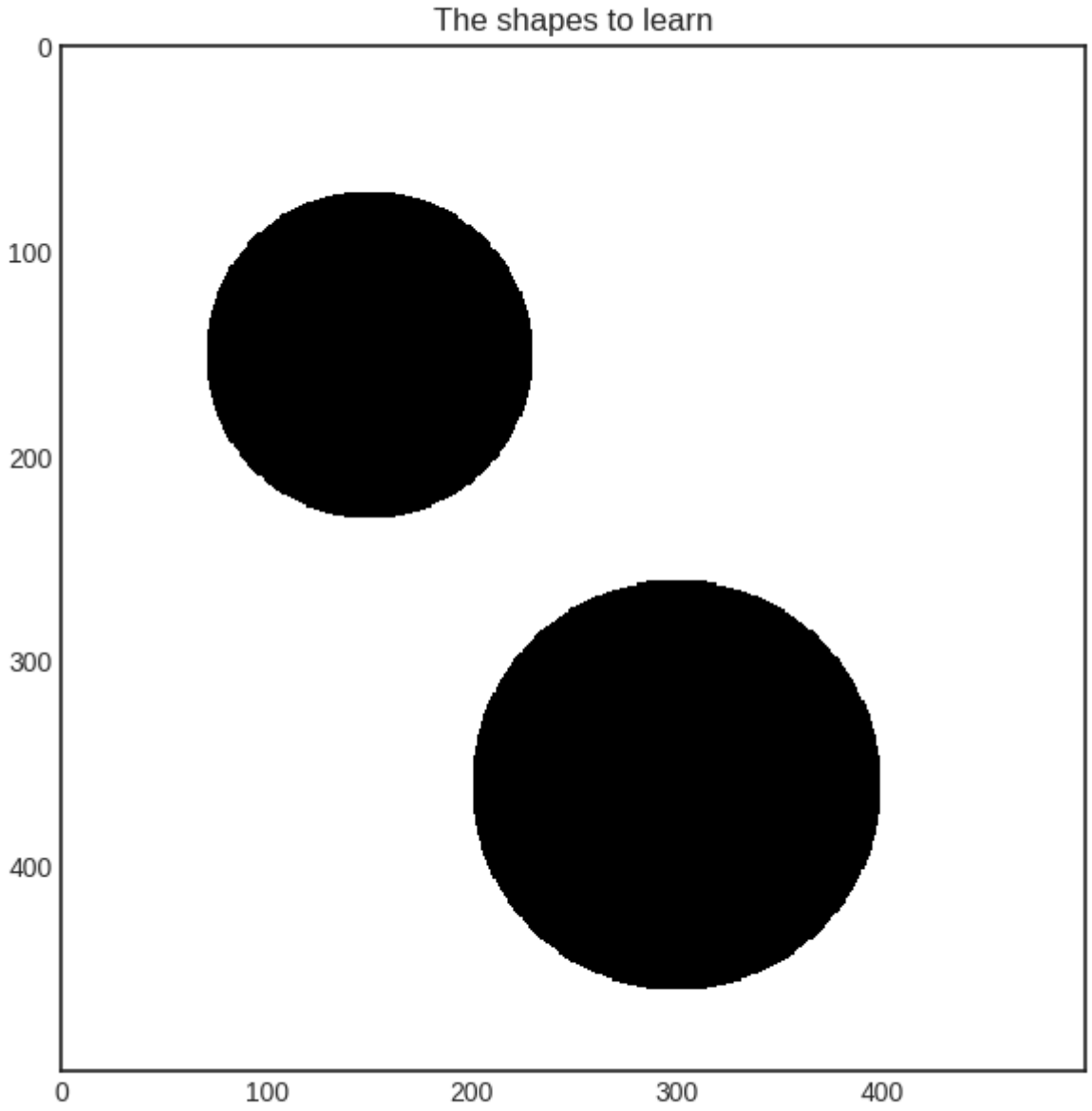
To visualize the disagreement measures, let's consider a toy example! Suppose that we would like to learn these two objects:

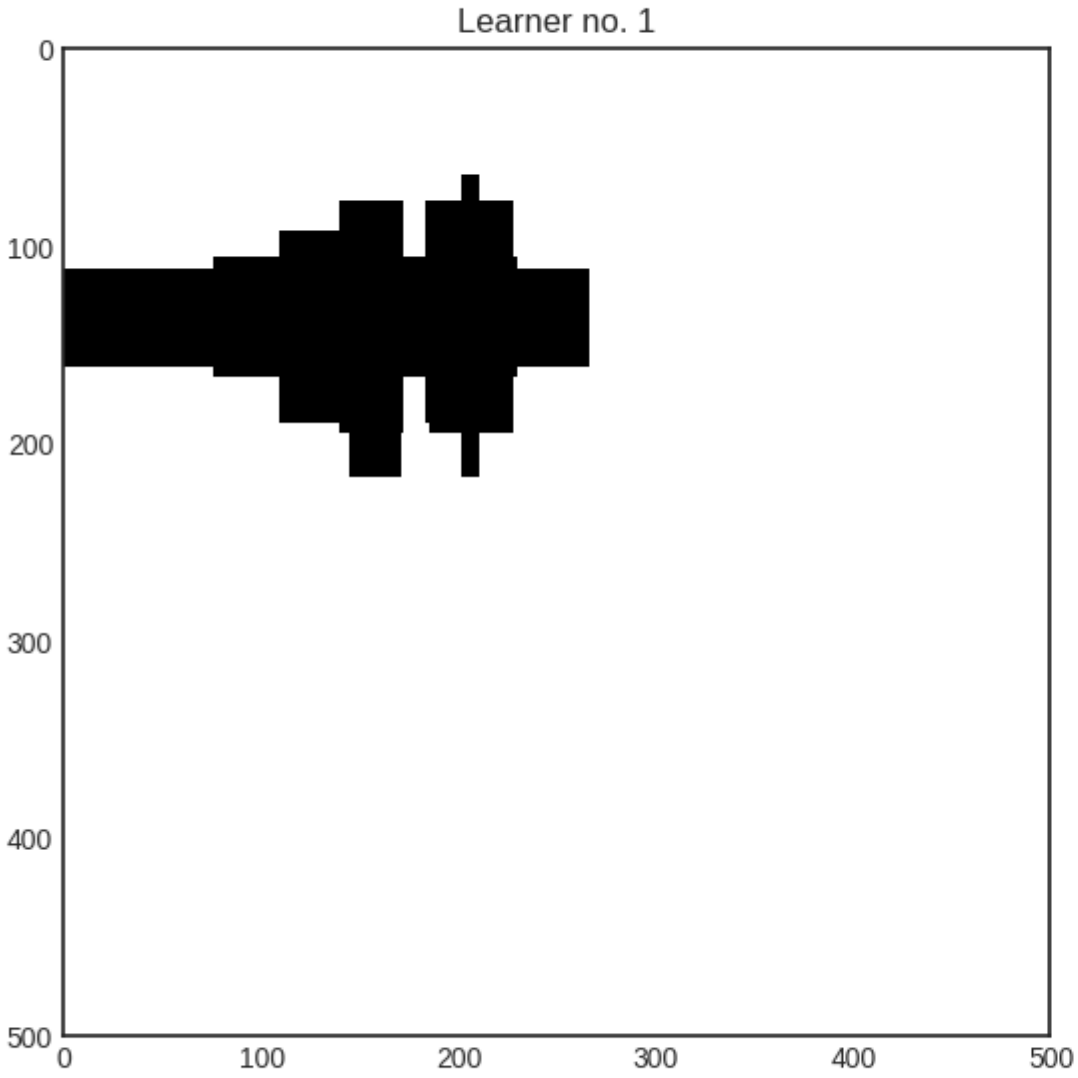
We train two random forest classifiers:

The consensus predictions of these learners are

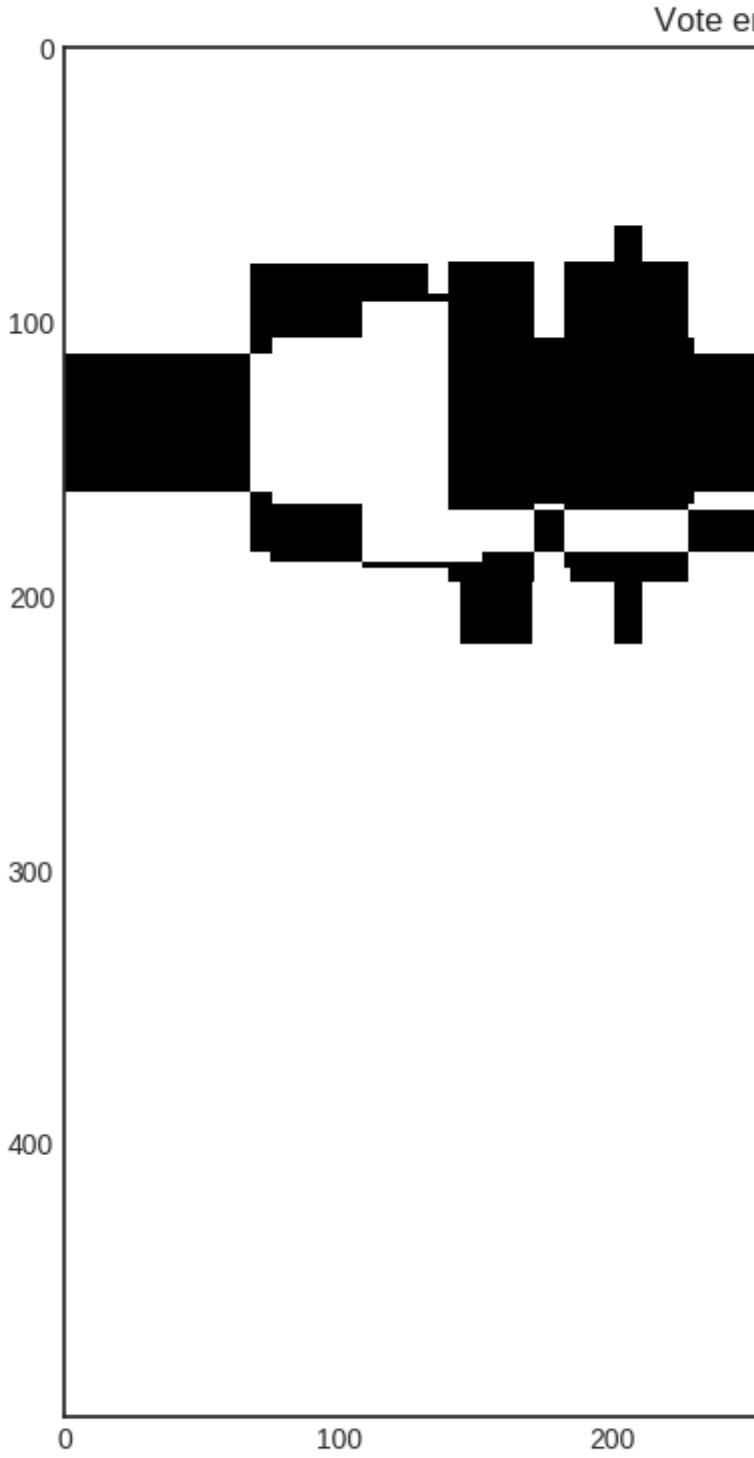
In this case, the disagreement measures from left to right are vote entropy, consensus entropy, and max disagreement.


Regressors
1.25
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00


The shapes to learn
0
100
200
300
400
0
100
200
300
400


Learner no. 1
0
100
200
300
400
500
0
100
200
300
400
500


Consensus
0
100
200
300
400
0
100
200


Vote e
0
100
200
300
400
0
100
200

CHAPTER 12

Ranked batch-mode sampling

When querying from unlabeled data, sometimes you have the resources to label multiple instances at the same time. Classical uncertainty sampling however, does not quite support this scenario. To address this issue, multiple strategies were proposed, for instance Ranked batch-mode queries by Cardoso et al., which is implemented in modAL.

In this method, each example x is scored using the formula

$$score = \alpha(1 - \Phi(x, X_{labeled})) + (1 - \alpha)U(x),$$

where $\alpha = \frac{|X_{unlabeled}|}{|X_{unlabeled}|+|X_{labeled}|}$, $X_{labeled}$ is the labeled dataset, $U(x)$ is the uncertainty of predictions for x, and Φ is a so-called similarity function, for instance cosine similarity. This latter function measures how well the feature space is explored near x. (The lower the better.)

After scoring, the highest scored instance is put at the top of a list. The instance is removed from the pool and the score is recalculated until the desired amount of instances are selected.

In here, we will use the Iris dataset to demonstrate this. For more information on the iris dataset, see its wikipedia page or the documentation of its scikit-learn interface.

[1]:
```
import numpy as np
from sklearn.datasets import load_iris

# Loading the dataset
iris = load_iris()
X_raw = iris['data']
y_raw = iris['target']

# Isolate our examples for our labeled dataset.
n_labeled_examples = X_raw.shape[0]
training_indices = np.random.randint(low=0, high=len(X_raw)+1, size=3)

# Defining the training data
X_training = X_raw[training_indices]
y_training = y_raw[training_indices]

# Isolate the non-training examples we'll be querying.
```

(continues on next page)

(continued from previous page)

```
X_pool = np.delete(X_raw, training_indices, axis=0)
y_pool = np.delete(y_raw, training_indices, axis=0)
```

For visualization purposes, we also perform a PCA transformation on the dataset.

[2]:
```
from sklearn.decomposition import PCA

# Define our PCA transformer and fit it onto our raw dataset.
pca = PCA(n_components=2)
pca.fit(X=X_raw)
```

[2]:
```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

Now we initialize an ActiveLearner.

[3]:
```
from modAL.models import ActiveLearner
from modAL.batch import uncertainty_batch_sampling
from sklearn.neighbors import KNeighborsClassifier

# Specify our core estimator.
knn = KNeighborsClassifier(n_neighbors=3)

learner = ActiveLearner(
    estimator=knn,
    query_strategy=uncertainty_batch_sampling,
    X_training=X_training, y_training=y_training
)
```

[4]:
```
from modAL.batch import ranked_batch
from modAL.uncertainty import classifier_uncertainty
from sklearn.metrics.pairwise import pairwise_distances

uncertainty = classifier_uncertainty(learner, X_pool)
distance_scores = pairwise_distances(X_pool, X_training, metric='euclidean').
↪min(axis=1)
similarity_scores = 1 / (1 + distance_scores)

alpha = len(X_pool)/len(X_raw)

scores = alpha * (1 - similarity_scores) + (1 - alpha) * uncertainty
```

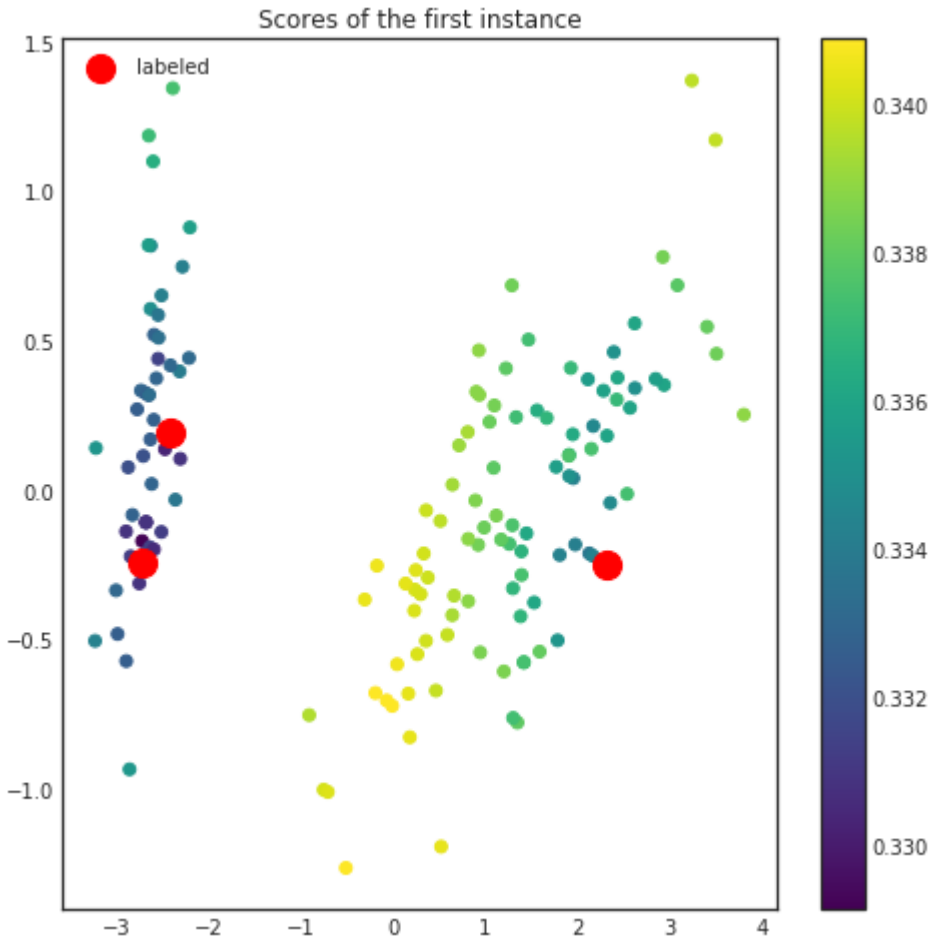
Now we can visualize the scores for the first instance during the first query.

[5]:
```
import matplotlib.pyplot as plt
%matplotlib inline
transformed_pool = pca.transform(X_pool)
transformed_training = pca.transform(X_training)

with plt.style.context('seaborn-white'):
    plt.figure(figsize=(8, 8))
    plt.scatter(transformed_pool[:, 0], transformed_pool[:, 1], c=scores, cmap=
↪'viridis')
    plt.colorbar()
    plt.scatter(transformed_training[:, 0], transformed_training[:, 1], c='r', s=200,␣
↪label='labeled')
    plt.title('Scores of the first instance')
```

(continues on next page)

(continued from previous page)

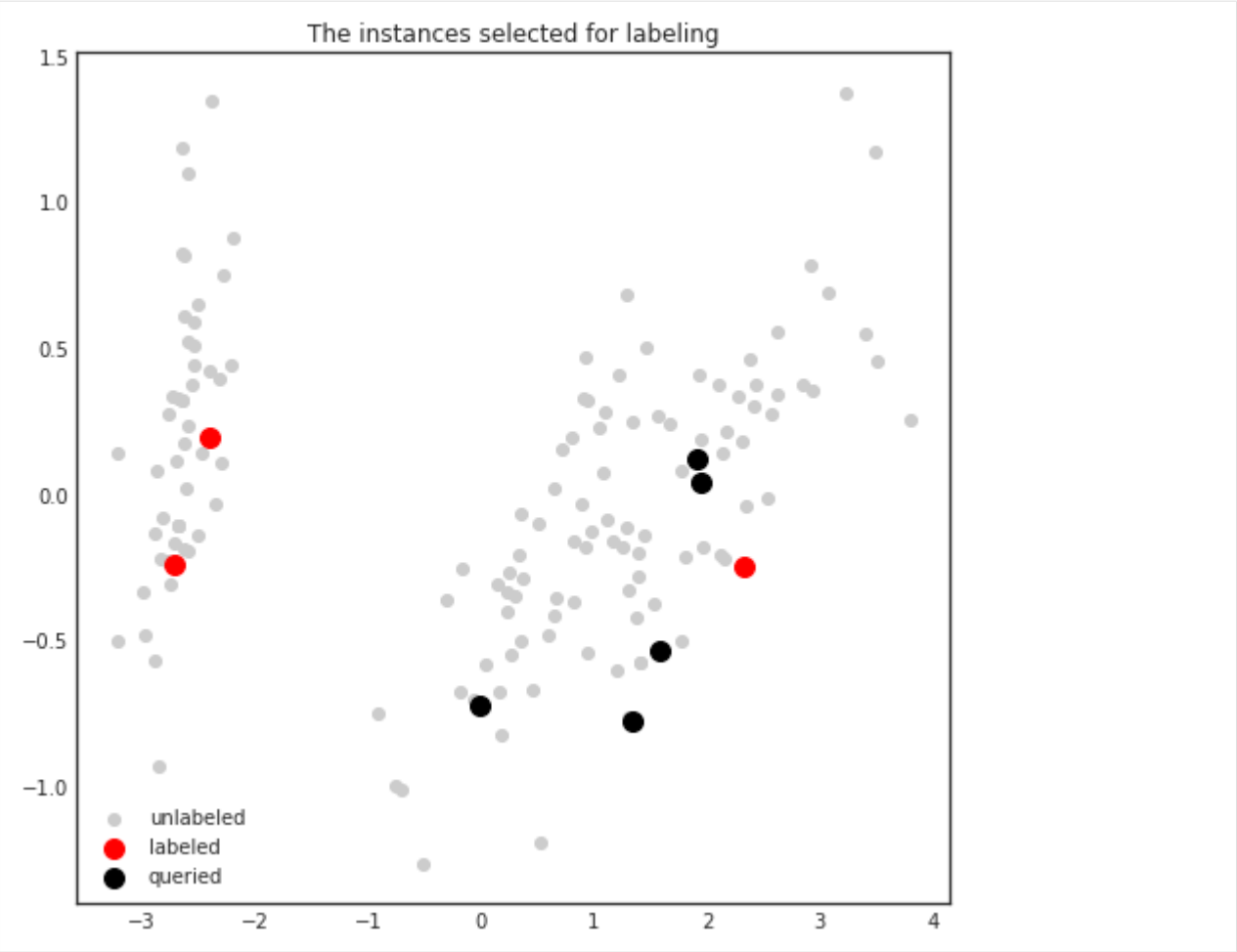
```
    plt.legend()
```



After the scores have been calculated, the highest scoring instance is selected and removed from the pool. **This instance is not queried yet!** The scores will be recalculated until the desired number of examples are available to send for query. In this setting, we shall visualize the instances to be queried in the first batch.

[6]:
```
query_idx, query_instances = learner.query(X_pool, n_instances=5)
```

[7]:
```
transformed_batch = pca.transform(query_instances)

with plt.style.context('seaborn-white'):
    plt.figure(figsize=(8, 8))
    plt.scatter(transformed_pool[:, 0], transformed_pool[:, 1], c='0.8', label=
↪'unlabeled')
    plt.scatter(transformed_training[:, 0], transformed_training[:, 1], c='r', s=100,␣
↪label='labeled')
    plt.scatter(transformed_batch[:, 0], transformed_batch[:, 1], c='k', s=100, label=
↪'queried')
    plt.title('The instances selected for labeling')
    plt.legend()
```


The instances selected for labeling
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4
unlabeled
labeled
queried

CHAPTER 13

Information density

When using uncertainty sampling (or other similar strategies), we are unable to take the structure of the data into account. This can lead us to suboptimal queries. To alleviate this, one method is to use information density measures to help us guide our queries.

For an unlabeled dataset X_u, the information density of an instance x can be calculated as

$$I(x) = \frac{1}{|X_u|} \sum_{x' \in X} sim(x, x'),$$

where $sim(x, x')$ is a similarity function such as cosine similarity or Euclidean similarity, which is the reciprocal of Euclidean distance. The higher the information density, the more similar the given instance is to the rest of the data. To illustrate this, we shall use a simple synthetic dataset.

For more details, see Section 5.1 of the Active Learning book by Burr Settles!

[1]:
```
from sklearn.datasets import make_blobs

X, y = make_blobs(n_features=2, n_samples=1000, centers=3, random_state=0, cluster_
↪std=0.7)
```

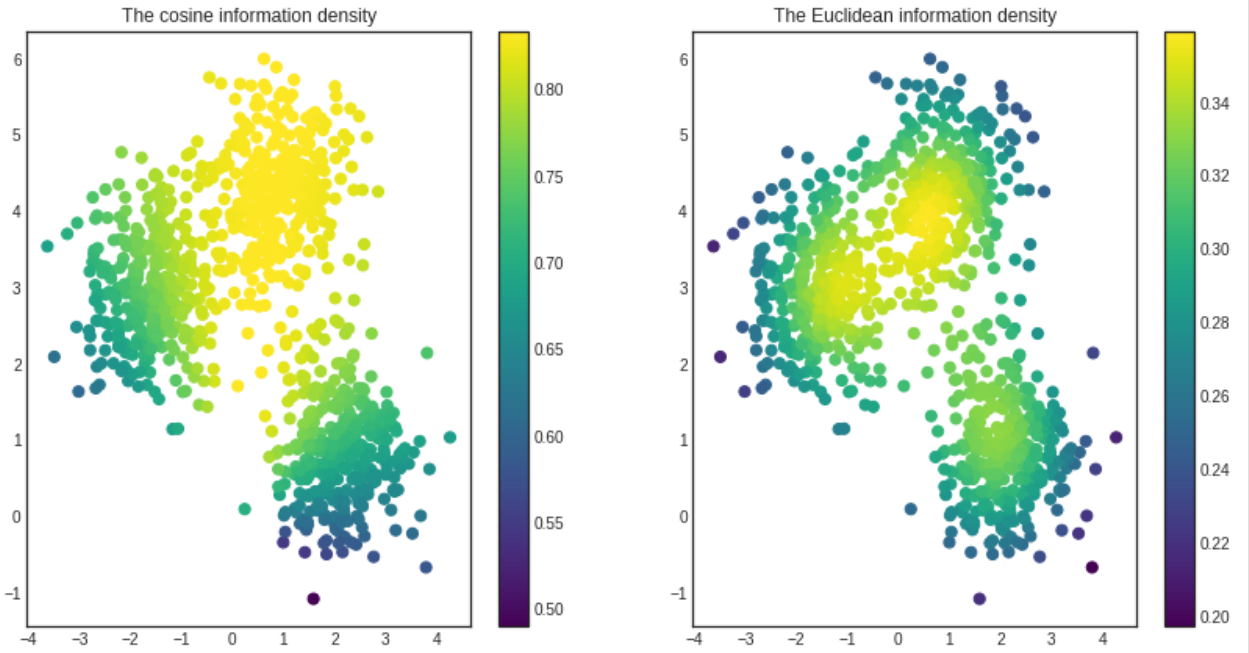
[2]:
```
from modAL.density import information_density

cosine_density = information_density(X, 'cosine')
euclidean_density = information_density(X, 'euclidean')
```

[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# visualizing the cosine and euclidean information density
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(14, 7))
    plt.subplot(1, 2, 1)
    plt.scatter(x=X[:, 0], y=X[:, 1], c=cosine_density, cmap='viridis', s=50)
    plt.title('The cosine information density')
```

(continues on next page)

(continued from previous page)

```
    plt.colorbar()
    plt.subplot(1, 2, 2)
    plt.scatter(x=X[:, 0], y=X[:, 1], c=euclidean_density, cmap='viridis', s=50)
    plt.title('The Euclidean information density')
    plt.colorbar()
    plt.show()
```



As you can see, the certain similarity functions highlight distinct features of the dataset. The Euclidean information density prefers the center of clusters, while the cosine describes the middle cluster as most important.

CHAPTER 14

Interactive labeling with Jupyter

In this example, the active learning workflow of modAL is demonstrated - with you in the loop! By running this notebook, you'll be queried to label digits using the DIGITS dataset. If you would like to try this out, you can download this notebook here!

[1]:
```
import numpy as np

from modAL.models import ActiveLearner
from modAL.uncertainty import uncertainty_sampling

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

from IPython import display
from matplotlib import pyplot as plt
%matplotlib inline
```

```
/home/namazu/.local/lib/python3.6/site-packages/sklearn/ensemble/weight_boosting.py:
↪29: DeprecationWarning: numpy.core.umath_tests is an internal NumPy module and␣
↪should not be imported. It will be removed in a future NumPy release.
  from numpy.core.umath_tests import inner1d
```

14.1 The dataset

Now we set up the initial training set for our classifier. If you would like to play around, you can try to modifiy the value `n_initial` below and see if it impacts the algorithm!

[2]:
```
n_initial = 100
```

[3]:
```
X, y = load_digits(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y)
```

(continues on next page)

(continued from previous page)

```
initial_idx = np.random.choice(range(len(X_train)), size=n_initial, replace=False)

X_initial, y_initial = X_train[initial_idx], y_train[initial_idx]
X_pool, y_pool = np.delete(X_train, initial_idx, axis=0), np.delete(y_train, initial_
↪idx, axis=0)
```

14.2 Initializing the learner

Now we initialize the active learner. Feel free to change the underlying `RandomForestClassifier` or the `uncertainty_sampling`!

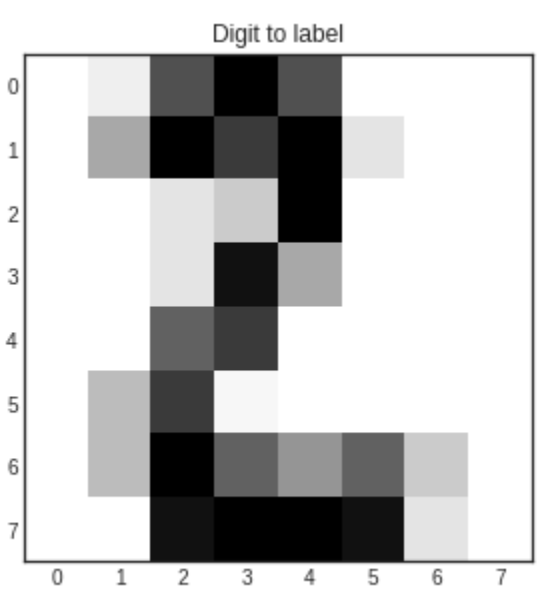
[4]:
```
learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    query_strategy=uncertainty_sampling,
    X_training=X_initial, y_training=y_initial
)
```

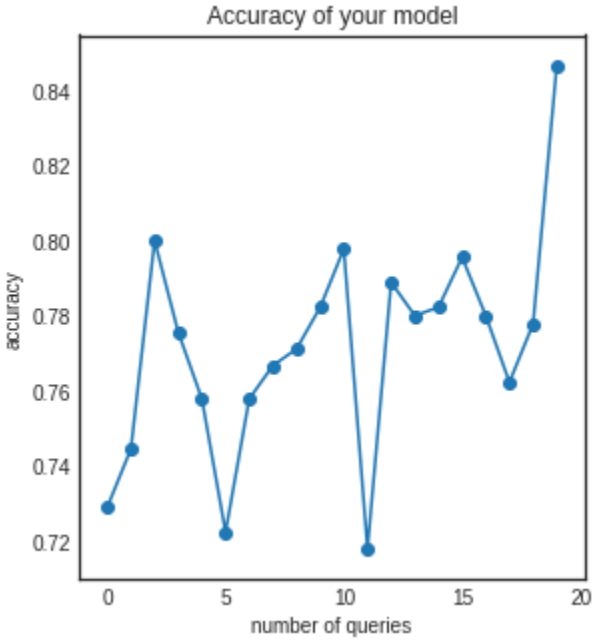
We also set how many queries we want to make. The more the better! (Usually :))

[5]:
```
n_queries = 20
```

14.3 The active learning loop

[6]:
```
accuracy_scores = [learner.score(X_test, y_test)]

for i in range(n_queries):
    display.clear_output(wait=True)
    query_idx, query_inst = learner.query(X_pool)
    with plt.style.context('seaborn-white'):
        plt.figure(figsize=(10, 5))
        plt.subplot(1, 2, 1)
        plt.title('Digit to label')
        plt.imshow(query_inst.reshape(8, 8))
        plt.subplot(1, 2, 2)
        plt.title('Accuracy of your model')
        plt.plot(range(i+1), accuracy_scores)
        plt.scatter(range(i+1), accuracy_scores)
        plt.xlabel('number of queries')
        plt.ylabel('accuracy')
        display.display(plt.gcf())
        plt.close('all')

    print("Which digit is this?")
    y_new = np.array([int(input())], dtype=int)
    learner.teach(query_inst.reshape(1, -1), y_new)
    X_pool, y_pool = np.delete(X_pool, query_idx, axis=0), np.delete(y_pool, query_
↪idx, axis=0)
    accuracy_scores.append(learner.score(X_test, y_test))
```

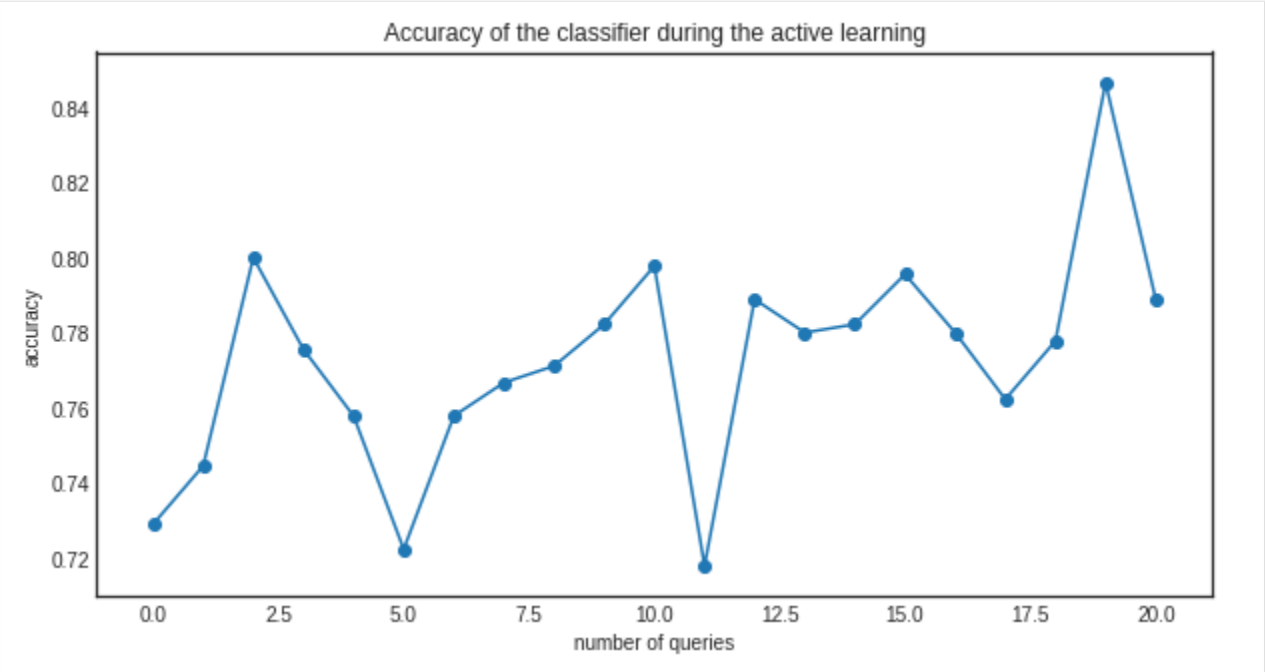





```
Which digit is this?
2
```

Finally, we can visualize the accuracy during the training.

[7]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 5))
    plt.title('Accuracy of the classifier during the active learning')
    plt.plot(range(n_queries+1), accuracy_scores)
    plt.scatter(range(n_queries+1), accuracy_scores)
    plt.xlabel('number of queries')
    plt.ylabel('accuracy')
    plt.show()
```


Accuracy of the classifier during the active learning
accuracy
0.84
0.82
0.80
0.78
0.76
0.74
0.72
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
number of queries

CHAPTER 15

Pool-based sampling

15.1 Overview

In this example, we apply an `ActiveLearner` onto the iris dataset using pool-based sampling. In this setting, we assume a small set of labeled data $\mathcal{L}$ and a large set of unlabeled data $\mathcal{U}$ such that $|\mathcal{L}| \ll |\mathcal{U}|$. In his review of the active learning literature, Settles covers a high-level overview of the general pool-based sampling algorithm:

> Queries are selectively drawn from the pool, which is usually assumed to be closed (i.e., static or non-changing), although this is not strictly necessary. Typically, instances are queried in a greedy fashion, according to an informativeness measure used to evaluate all instances in the pool (or, perhaps if $\mathcal{U}$ is very large, some subsample thereof).

Along with our pool-based sampling strategy, `modAL`'s modular design allows you to vary parameters surrounding the active learning process, including the core estimator and query strategy. In this example, we use `scikit-learn`'s k-nearest neighbors classifier as our estimator and default to `modAL`'s uncertainty sampling query strategy.

For further reading on pool-based sampling, we highly recommend the following resources: - Burr Settles. Active Learning Literature Survey [**Section 2.3: Pool-based Sampling**]. Computer Sciences Technical Report 1648, University of Wisconsin-Madison. 2009.

To enforce a reproducible result across runs, we set a random seed.

[1]:
```
import numpy as np

# Set our RNG seed for reproducibility.
RANDOM_STATE_SEED = 123
np.random.seed(RANDOM_STATE_SEED)
```

15.2 The dataset

Now we load the dataset. In this example, we are going to use the famous Iris dataset. For more information on the iris dataset, see: - The dataset documentation on Wikipedia - The scikit-learn interface

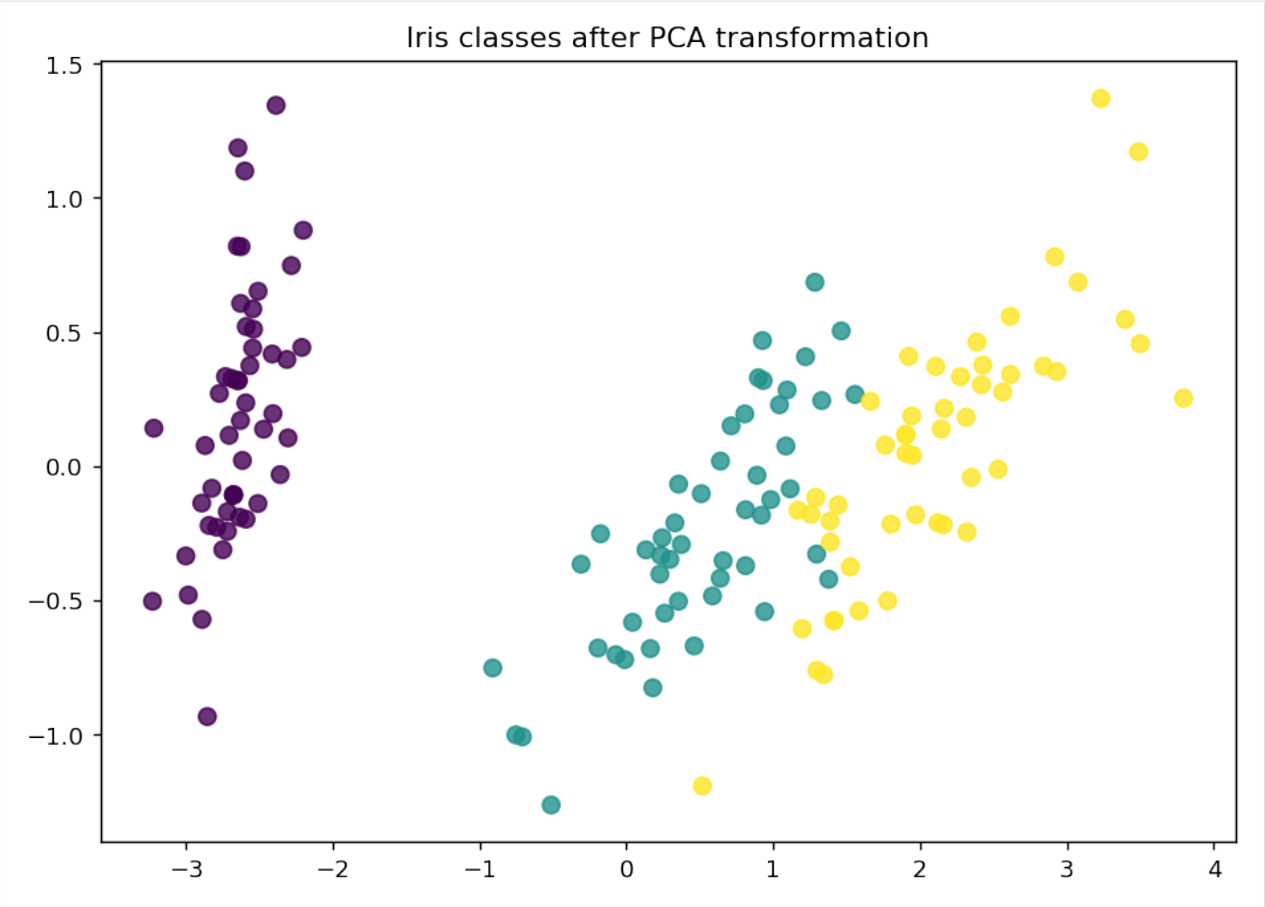
[2]:
```
from sklearn.datasets import load_iris

iris = load_iris()
X_raw = iris['data']
y_raw = iris['target']
```

For visualization purposes, we apply PCA to the original dataset.

[3]:
```
from sklearn.decomposition import PCA

# Define our PCA transformer and fit it onto our raw dataset.
pca = PCA(n_components=2, random_state=RANDOM_STATE_SEED)
transformed_iris = pca.fit_transform(X=X_raw)
```

This is how the dataset looks like.

[4]:
```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

# Isolate the data we'll need for plotting.
x_component, y_component = transformed_iris[:, 0], transformed_iris[:, 1]

# Plot our dimensionality-reduced (via PCA) dataset.
plt.figure(figsize=(8.5, 6), dpi=130)
plt.scatter(x=x_component, y=y_component, c=y_raw, cmap='viridis', s=50, alpha=8/10)
plt.title('Iris classes after PCA transformation')
plt.show()
```



Now we partition our `iris` dataset into a training set $\mathcal{L}$ and $\mathcal{U}$. We first specify our training set $\mathcal{L}$ consisting of 3 random examples. The remaining examples go to our "unlabeled" pool $\mathcal{U}$.

[5]:
```
# Isolate our examples for our labeled dataset.
n_labeled_examples = X_raw.shape[0]
training_indices = np.random.randint(low=0, high=n_labeled_examples + 1, size=3)

X_train = X_raw[training_indices]
y_train = y_raw[training_indices]

# Isolate the non-training examples we'll be querying.
X_pool = np.delete(X_raw, training_indices, axis=0)
y_pool = np.delete(y_raw, training_indices, axis=0)
```

15.3 Active learning with pool-based sampling

For the classification, we are going to use a simple k-nearest neighbors classifier. In this step, we are also going to initialize the `ActiveLearner`.

[6]:
```
from sklearn.neighbors import KNeighborsClassifier
from modAL.models import ActiveLearner

# Specify our core estimator along with it's active learning model.
```

(continues on next page)

(continued from previous page)

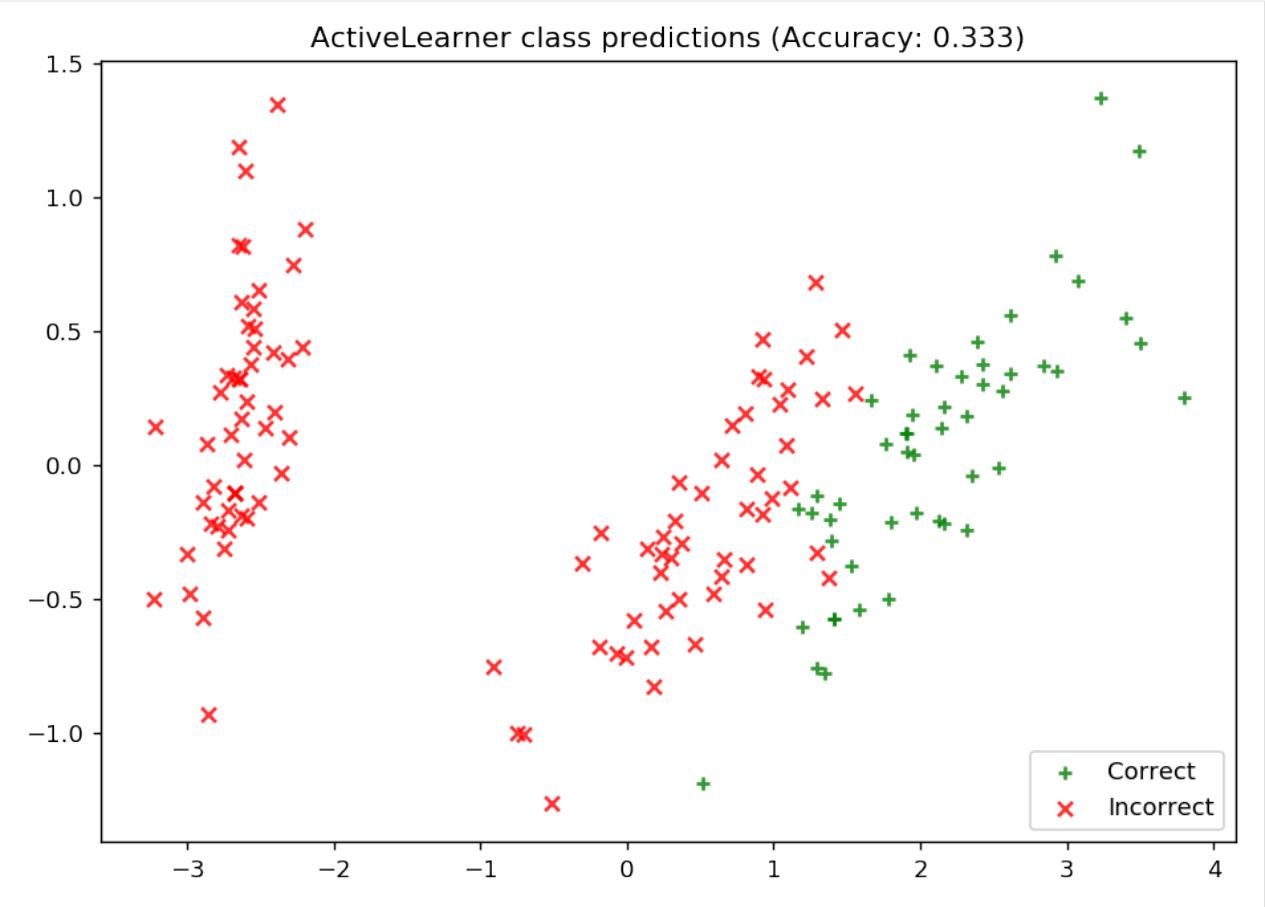
```
knn = KNeighborsClassifier(n_neighbors=3)
learner = ActiveLearner(estimator=knn, X_training=X_train, y_training=y_train)
```

Let's see how our classifier performs on the initial training set!

[7]:
```
# Isolate the data we'll need for plotting.
predictions = learner.predict(X_raw)
is_correct = (predictions == y_raw)

predictions
```

[7]:
```
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

[8]:
```
# Record our learner's score on the raw data.
unqueried_score = learner.score(X_raw, y_raw)

# Plot our classification results.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)
ax.scatter(x=x_component[is_correct],  y=y_component[is_correct],  c='g', marker='+',␣
↪label='Correct',   alpha=8/10)
ax.scatter(x=x_component[~is_correct], y=y_component[~is_correct], c='r', marker='x',␣
↪label='Incorrect', alpha=8/10)
ax.legend(loc='lower right')
ax.set_title("ActiveLearner class predictions (Accuracy: {score:.3f})".
↪format(score=unqueried_score))
plt.show()
```



15.4 Update our model by pool-based sampling our "unlabeled" dataset $\mathcal{U}$

As we can see, our model is unable to properly learn the underlying data distribution. All of its predictions are for the third class label, and as such it is only as competitive as defaulting its predictions to a single class – if only we had more data!

Below, we tune our classifier by allowing it to query 20 instances it hasn't seen before. Using uncertainty sampling, our classifier aims to reduce the amount of uncertainty in its predictions using a variety of measures — see the documentation for more on specific classification uncertainty measures. With each requested query, we remove that record from our pool $\mathcal{U}$ and record our model's accuracy on the raw dataset.

[9]:
```
N_QUERIES = 20
performance_history = [unqueried_score]

# Allow our model to query our unlabeled dataset for the most
# informative points according to our query strategy (uncertainty sampling).
for index in range(N_QUERIES):
  query_index, query_instance = learner.query(X_pool)

  # Teach our ActiveLearner model the record it has requested.
  X, y = X_pool[query_index].reshape(1, -1), y_pool[query_index].reshape(1, )
  learner.teach(X=X, y=y)
```

(continues on next page)

(continued from previous page)

```
  # Remove the queried instance from the unlabeled pool.
  X_pool, y_pool = np.delete(X_pool, query_index, axis=0), np.delete(y_pool, query_
↪index)

  # Calculate and report our model's accuracy.
  model_accuracy = learner.score(X_raw, y_raw)
  print('Accuracy after query {n}: {acc:0.4f}'.format(n=index + 1, acc=model_
↪accuracy))

  # Save our model's performance for plotting.
  performance_history.append(model_accuracy)
```

```
Accuracy after query 1: 0.6667
Accuracy after query 2: 0.6667
Accuracy after query 3: 0.8800
Accuracy after query 4: 0.8800
Accuracy after query 5: 0.8733
Accuracy after query 6: 0.8400
Accuracy after query 7: 0.7400
Accuracy after query 8: 0.7267
Accuracy after query 9: 0.7267
Accuracy after query 10: 0.7267
Accuracy after query 11: 0.7267
Accuracy after query 12: 0.7267
Accuracy after query 13: 0.7267
Accuracy after query 14: 0.7267
Accuracy after query 15: 0.7200
Accuracy after query 16: 0.8400
Accuracy after query 17: 0.8800
Accuracy after query 18: 0.8933
Accuracy after query 19: 0.9267
Accuracy after query 20: 0.9267
```

15.5 Evaluate our model's performance

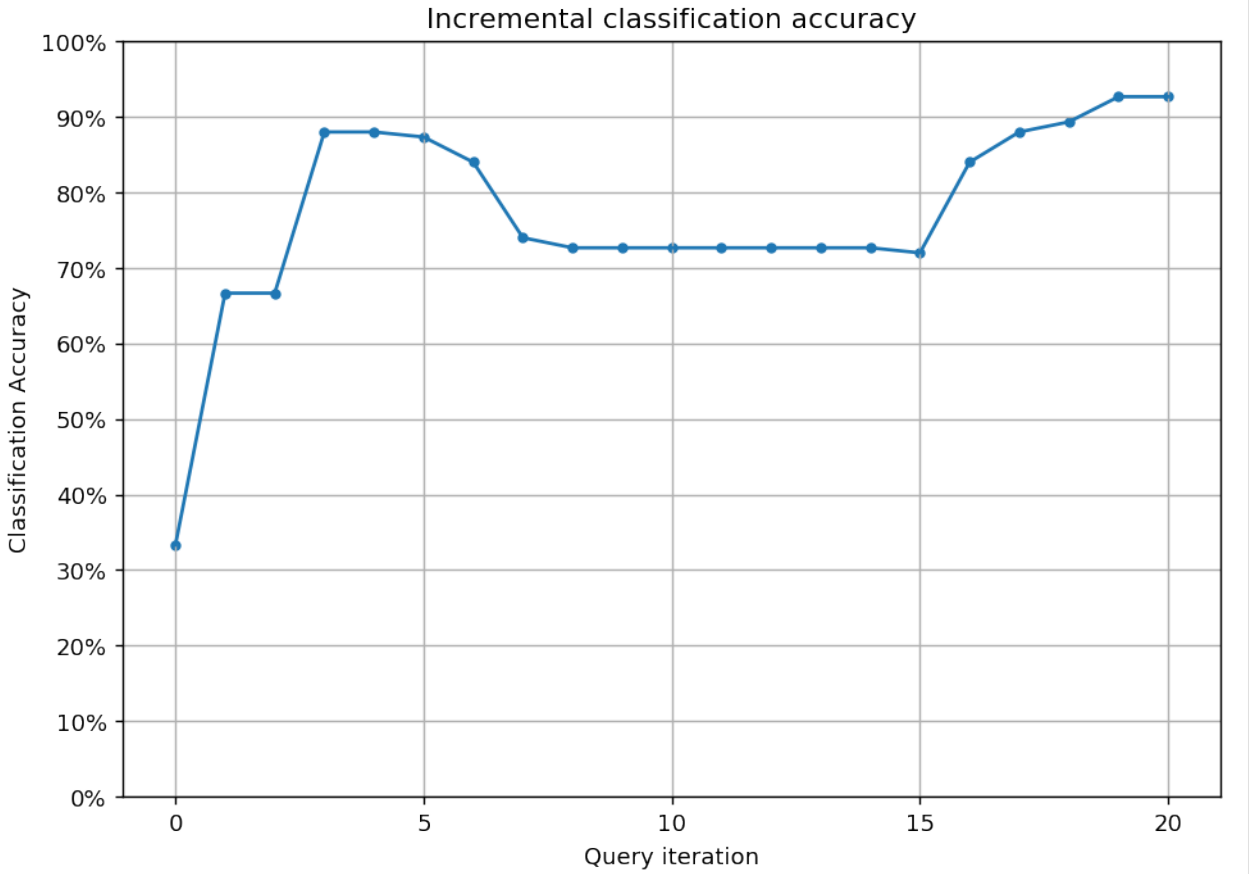
Here, we first plot the query iteration index against model accuracy. To visualize the performance of our classifier, we also plot the correct and incorrect predictions on the full dataset.

[10]:
```
# Plot our performance over time.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)

ax.plot(performance_history)
ax.scatter(range(len(performance_history)), performance_history, s=13)

ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=5, integer=True))
ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=10))
ax.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(xmax=1))

ax.set_ylim(bottom=0, top=1)
ax.grid(True)

ax.set_title('Incremental classification accuracy')
ax.set_xlabel('Query iteration')
```

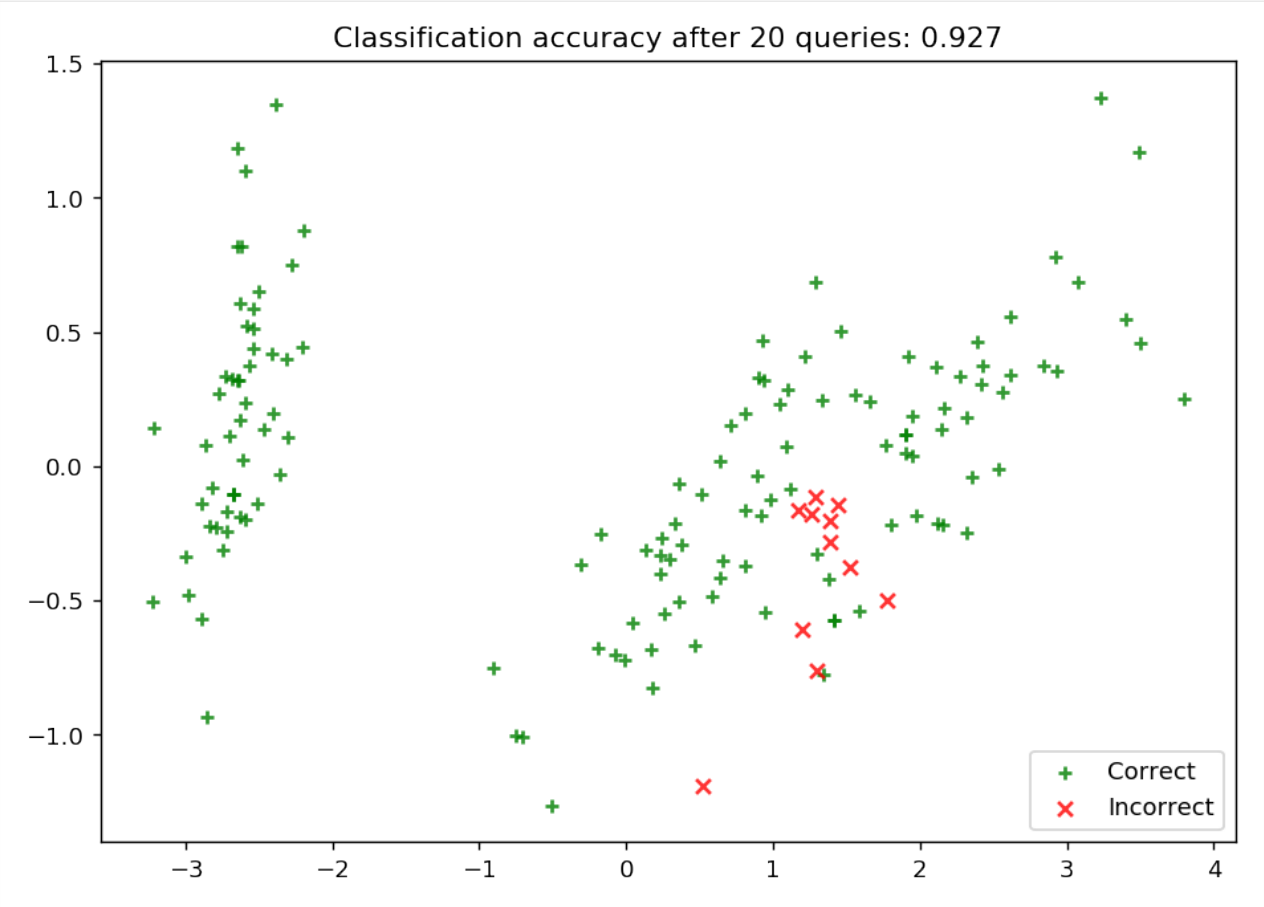
(continues on next page)

(continued from previous page)

```
ax.set_ylabel('Classification Accuracy')

plt.show()
```



[11]:
```
# Isolate the data we'll need for plotting.
predictions = learner.predict(X_raw)
is_correct = (predictions == y_raw)

# Plot our updated classification results once we've trained our learner.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)

ax.scatter(x=x_component[is_correct],  y=y_component[is_correct],  c='g', marker='+',␣
↪label='Correct',   alpha=8/10)
ax.scatter(x=x_component[~is_correct], y=y_component[~is_correct], c='r', marker='x',␣
↪label='Incorrect', alpha=8/10)

ax.set_title('Classification accuracy after {n} queries: {final_acc:.3f}'.format(n=N_
↪QUERIES, final_acc=performance_history[-1]))
ax.legend(loc='lower right')

plt.show()
```


Classification accuracy after 20 queries: 0.927
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4
Correct
Incorrect

CHAPTER 16

Ranked batch-mode sampling

16.1 Overview

In this example, we apply an `ActiveLearner` onto the iris dataset using **ranked batch-mode active learning**. Much like pool-based sampling, in this setting we assume a small (potentially empty) set of labeled data $\mathcal{L}$ and a large set of unlabeled data $\mathcal{U}$ such that $|\mathcal{L}| \ll |\mathcal{U}|$.

One of the drawbacks of standard pool-based sampling (indeed, most sampling techniques) is their interactive nature: these sampling methods are best fit for cases where we have an attentive human-in-the-loop for maximizing the model's performance. Although these sampling methods (ours included!) *do* support returning multiple samples per query, they tend to return redundant/sub-optimal queries if we return more than one instance from the unlabeled set. This is a bit prohibitive in settings where we'd like to ask an active learner to return multiple (if not all) examples from the unlabeled set/pool.

This implementation (see the original issue here) implements Cardoso et al.'s **ranked batch-mode** sampling, which not only supports batch-mode sampling (sampling methods that are built with querying multiple labels from the unlabeled set) but also establishes a *ranking* among the batch in order for end-users to prioritize which records to label from the unlabeled set. In their own words:

> Our new approach allows the algorithm to generate an arbitrarily long query, thus making its execution less frequent. For example, a ranked query containing every available instance could be generated outside working hours. This would allow hired analysts to label instances for a full day, in parallel, if desired, without waiting for the learner update and query reconstruction.

For further reading on ranked batch-mode active learning, we highly recommend the following resources:

- Thiago N.C. Cardoso, Rodrigo M. Silva, Sérgio Canuto, Mirella M. Moro, Marcos A. Gonçalves. Ranked batch-mode active learning. Information Sciences, Volume 379, 2017, Pages 313-337.

16.2 Note

Although this notebook only highlights the benefits of batch sampling (as opposed to interactive sampling), the same ideas hold for ranked batch-mode sampling. One day we'll get around to producing examples where ranked batch-

mode sampling is the core idea being evaluated :)

To enforce a reproducible result across runs, we set a random seed.

[1]:
```
import numpy as np

# Set our RNG for reproducibility.
RANDOM_STATE_SEED = 123
np.random.seed(RANDOM_STATE_SEED)
```

16.3 The dataset

Now we load the dataset. In this example, we are going to use the famous Iris dataset. For more information on the iris dataset, see: - The dataset documentation on Wikipedia - The scikit-learn interface

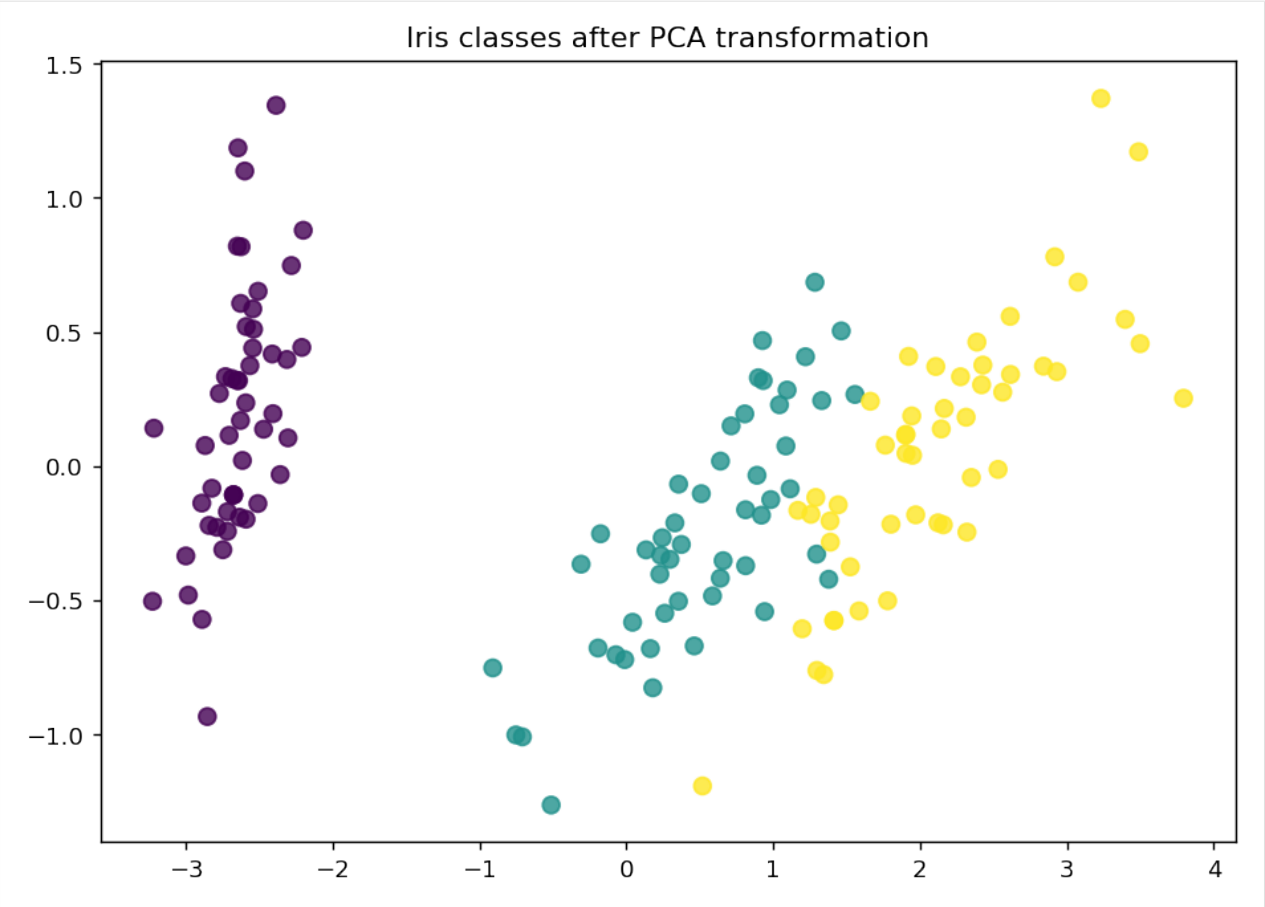
[2]:
```
from sklearn.datasets import load_iris

iris = load_iris()
X_raw = iris['data']
y_raw = iris['target']
```

For visualization purposes, we apply PCA to the original dataset.

[3]:
```
from sklearn.decomposition import PCA

# Define our PCA transformer and fit it onto our raw dataset.
pca = PCA(n_components=2, random_state=RANDOM_STATE_SEED)
transformed_iris = pca.fit_transform(X=X_raw)
```

This is how the dataset looks like.

[4]:
```
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt

# Isolate the data we'll need for plotting.
x_component, y_component = transformed_iris[:, 0], transformed_iris[:, 1]

# Plot our dimensionality-reduced (via PCA) dataset.
plt.figure(figsize=(8.5, 6), dpi=130)
plt.scatter(x=x_component, y=y_component, c=y_raw, cmap='viridis', s=50, alpha=8/10)
plt.title('Iris classes after PCA transformation')
plt.show()
```



Now we partition our `iris` dataset into a training set $\mathcal{L}$ and $\mathcal{U}$. We first specify our training set $\mathcal{L}$ consisting of 3 random examples. The remaining examples go to our "unlabeled" pool $\mathcal{U}$.

[5]:
```
# Isolate our examples for our labeled dataset.
n_labeled_examples = X_raw.shape[0]
training_indices = np.random.randint(low=0, high=n_labeled_examples + 1, size=3)

X_train = X_raw[training_indices]
y_train = y_raw[training_indices]

# Isolate the non-training examples we'll be querying.
X_pool = np.delete(X_raw, training_indices, axis=0)
y_pool = np.delete(y_raw, training_indices, axis=0)
```

16.4 Active learning with ranked batch mode sampling

For the classification, we are going to use a simple k-nearest neighbors classifier.

[6]:
```
from sklearn.neighbors import KNeighborsClassifier

# Specify our core estimator.
knn = KNeighborsClassifier(n_neighbors=3)
```

Now we initialize the `ActiveLearner`.

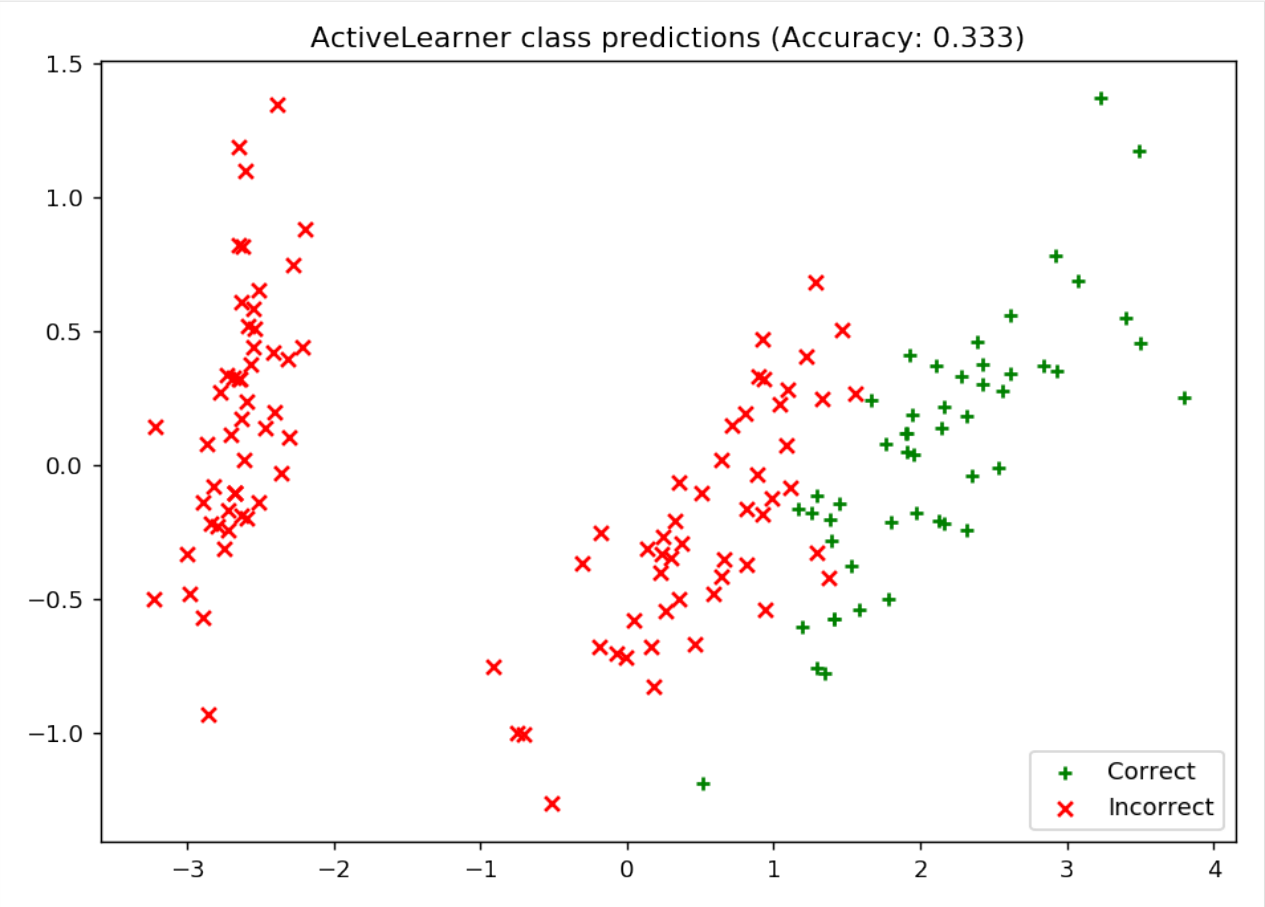
[7]:
```
from functools import partial
from modAL.batch import uncertainty_batch_sampling
from modAL.models import ActiveLearner

# Pre-set our batch sampling to retrieve 3 samples at a time.
BATCH_SIZE = 3
preset_batch = partial(uncertainty_batch_sampling, n_instances=BATCH_SIZE)

# Specify our active learning model.
learner = ActiveLearner(
  estimator=knn,

  X_training=X_train,
  y_training=y_train,

  query_strategy=preset_batch
)
```

Let's see how our classifier performs on the initial training set!

[8]:
```
# Isolate the data we'll need for plotting.
predictions = learner.predict(X_raw)
is_correct = (predictions == y_raw)

predictions
```

[8]:
```
array([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

[9]:
```
# Record our learner's score on the raw data.
unqueried_score = learner.score(X_raw, y_raw)

# Plot our classification results.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)
ax.scatter(x=x_component[is_correct],  y=y_component[is_correct],  c='g', marker='+',␣
↪label='Correct')
ax.scatter(x=x_component[~is_correct], y=y_component[~is_correct], c='r', marker='x',␣
↪label='Incorrect')
ax.legend(loc='lower right')
ax.set_title("ActiveLearner class predictions (Accuracy: {score:.3f})".
↪format(score=unqueried_score))
plt.show()
```



Now we Update our model by batch-mode sampling our "unlabeled" dataset $\mathcal{U}$. We tune our classifier by allowing it to query at most 20 instances it hasn't seen before. To properly utilize batch-mode sampling, we allow our model to request three records per query (instead of 1) but subsequently only allow our model to make 6 queries. Under the hood, our classifier aims to balance the ideas behind uncertainty and dissimilarity in its choices.

With each requested query, we remove that record from our pool $\mathcal{U}$ and record our model's accuracy on the raw dataset.

[10]:
```
# Pool-based sampling
N_RAW_SAMPLES = 20
N_QUERIES = N_RAW_SAMPLES // BATCH_SIZE

performance_history = [unqueried_score]

for index in range(N_QUERIES):
    query_index, query_instance = learner.query(X_pool)

    # Teach our ActiveLearner model the record it has requested.
    X, y = X_pool[query_index], y_pool[query_index]
    learner.teach(X=X, y=y)

    # Remove the queried instance from the unlabeled pool.
    X_pool = np.delete(X_pool, query_index, axis=0)
    y_pool = np.delete(y_pool, query_index)

    # Calculate and report our model's accuracy.
    model_accuracy = learner.score(X_raw, y_raw)
```

(continues on next page)

(continued from previous page)

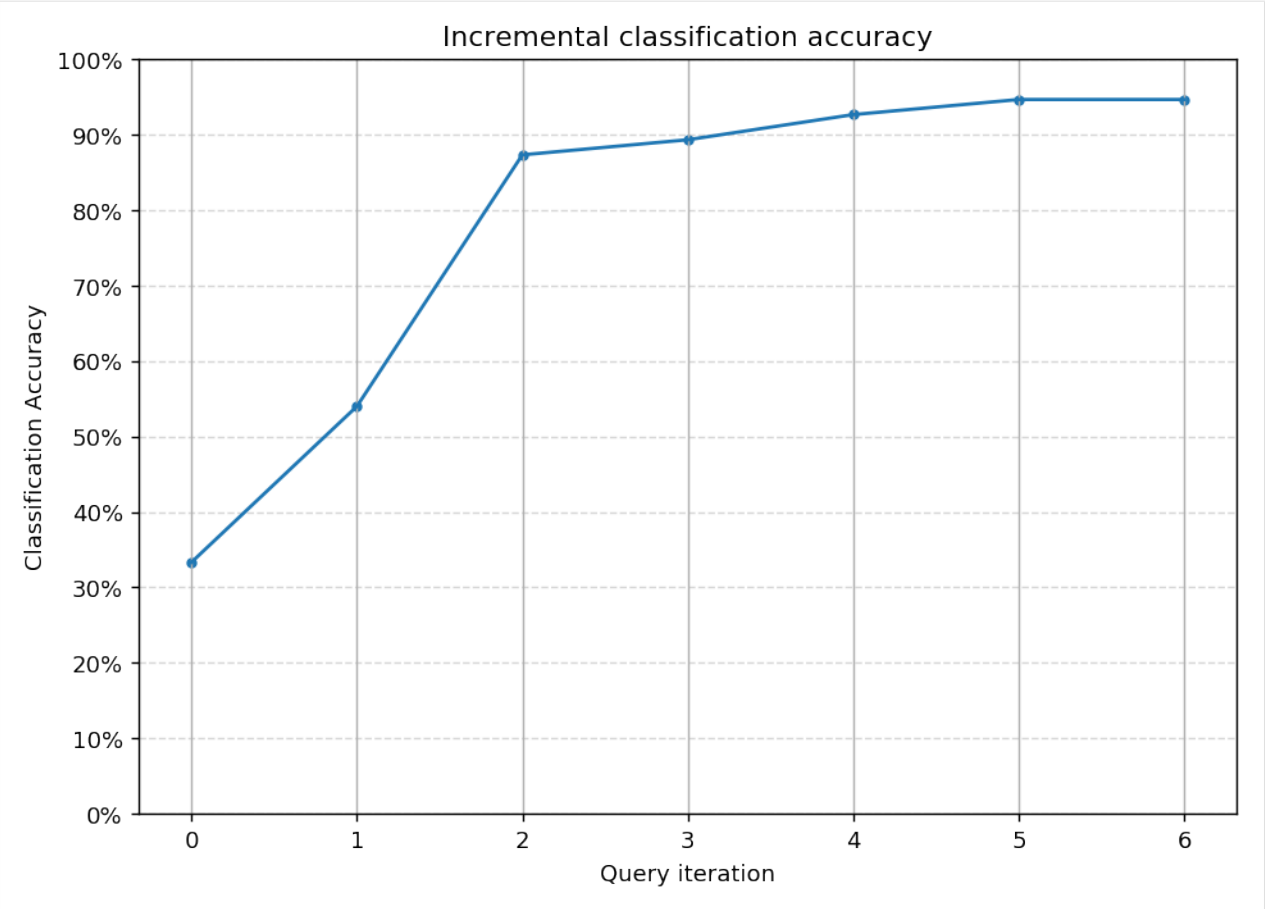
```
    print('Accuracy after query {n}: {acc:0.4f}'.format(n=index + 1, acc=model_
↪accuracy))

    # Save our model's performance for plotting.
    performance_history.append(model_accuracy)
```

```
Accuracy after query 1: 0.5400
Accuracy after query 2: 0.8733
Accuracy after query 3: 0.8933
Accuracy after query 4: 0.9267
Accuracy after query 5: 0.9467
Accuracy after query 6: 0.9467
```

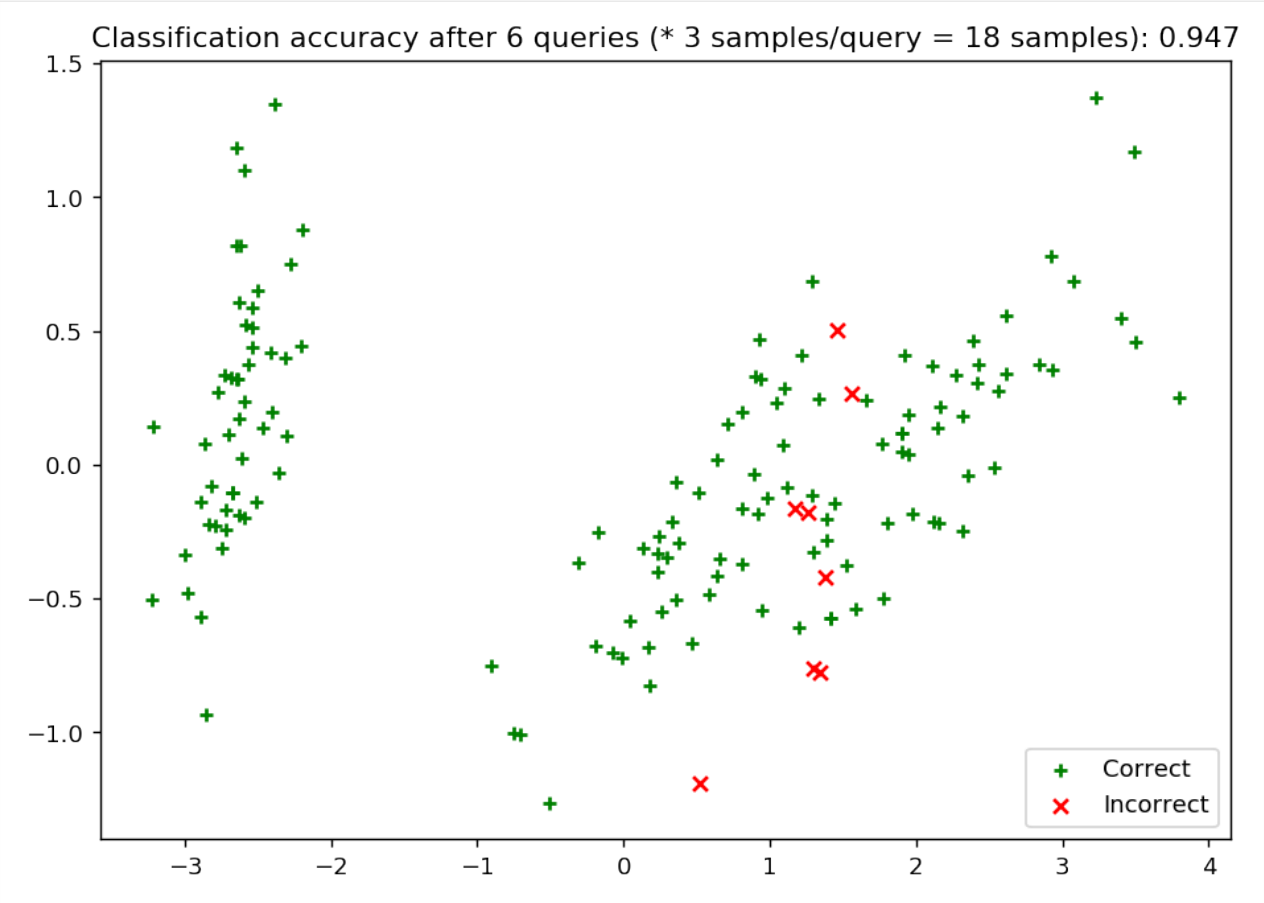
16.5 Evaluate our model's performance

Here, we first plot the query iteration index against model accuracy. As you can see, our model is able to obtain an accuracy of ~0.90 within its first query, and isn't as susceptible to getting "stuck" with querying uninformative records from our unlabeled set. To visualize the performance of our classifier, we also plot the correct and incorrect predictions on the full dataset.

[11]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt

# Plot our performance over time.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)

ax.plot(performance_history)
ax.scatter(range(len(performance_history)), performance_history, s=13)

ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=N_QUERIES + 3, integer=True))
ax.xaxis.grid(True)

ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=10))
ax.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(xmax=1))
ax.set_ylim(bottom=0, top=1)
ax.yaxis.grid(True, linestyle='--', alpha=1/2)

ax.set_title('Incremental classification accuracy')
ax.set_xlabel('Query iteration')
ax.set_ylabel('Classification Accuracy')

plt.show()
```



[12]:
```
# Isolate the data we'll need for plotting.
predictions = learner.predict(X_raw)
is_correct = (predictions == y_raw)

# Plot our updated classification results once we've trained our learner.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)

ax.scatter(x=x_component[is_correct],  y=y_component[is_correct],  c='g', marker='+',␣
↪label='Correct')
ax.scatter(x=x_component[~is_correct], y=y_component[~is_correct], c='r', marker='x',␣
↪label='Incorrect')

ax.set_title('Classification accuracy after {n} queries (* {batch_size} samples/query␣
↪= {total} samples): {final_acc:.3f}'.format(
    n=N_QUERIES,
    batch_size=BATCH_SIZE,
    total=N_QUERIES * BATCH_SIZE,
    final_acc=performance_history[-1]
))
ax.legend(loc='lower right')

plt.show()
```


Classification accuracy after 6 queries (* 3 samples/query = 18 samples): 0.947
1.5
1.0
0.5
0.0
−0.5
−1.0
−3
−2
−1
0
1
2
3
4
Correct
Incorrect

CHAPTER 17

Stream-based sampling

In addition to pool-based sampling, the stream-based scenario can also be implemented easily with modAL. In this case, the labels are not queried from a pool of instances. Rather, they are given one-by-one for the learner, which queries for its label if it finds the example useful. For instance, an example can be marked as useful if the prediction is uncertain, because acquiring its label would remove this uncertainty.

The executable script for this example can be found here!

To enforce a reproducible result across runs, we set a random seed.

17.1 The dataset

In this example, we are going to learn a black square on a white background. We are going to use a random forest classifier in a stream-based setting. First, let's generate some data!

[1]:
```
import numpy as np

# creating the image
im_width = 500
im_height = 500
im = np.zeros((im_height, im_width))
im[100:im_width - 1 - 100, 100:im_height - 1 - 100] = 1

# create the data to stream from
X_full = np.transpose(
    [np.tile(np.asarray(range(im.shape[0])), im.shape[1]),
     np.repeat(np.asarray(range(im.shape[1])), im.shape[0])]
)
# map the intensity values against the grid
y_full = np.asarray([im[P[0], P[1]] for P in X_full])

# create the data to stream from
X_full = np.transpose(
```

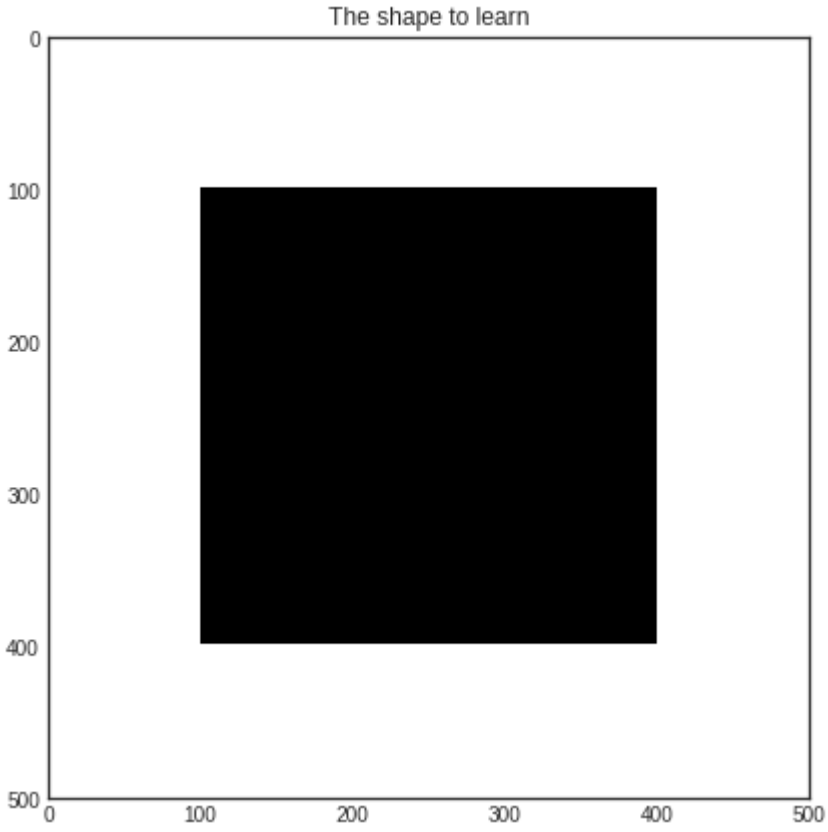
(continues on next page)

(continued from previous page)

```
    [np.tile(np.asarray(range(im.shape[0])), im.shape[1]),
     np.repeat(np.asarray(range(im.shape[1])), im.shape[0])]
)
# map the intensity values against the grid
y_full = np.asarray([im[P[0], P[1]] for P in X_full])
```

In case you are wondering, here is how this looks like!

[2]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline

with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    plt.imshow(im)
    plt.title('The shape to learn')
    plt.show()
```



17.2 Active learning with stream-based sampling

For classification, we will use a random forest classifier. Initializing the learner is the same as always.

[3]:
```
from sklearn.ensemble import RandomForestClassifier
from modAL.models import ActiveLearner
```

(continues on next page)

(continued from previous page)

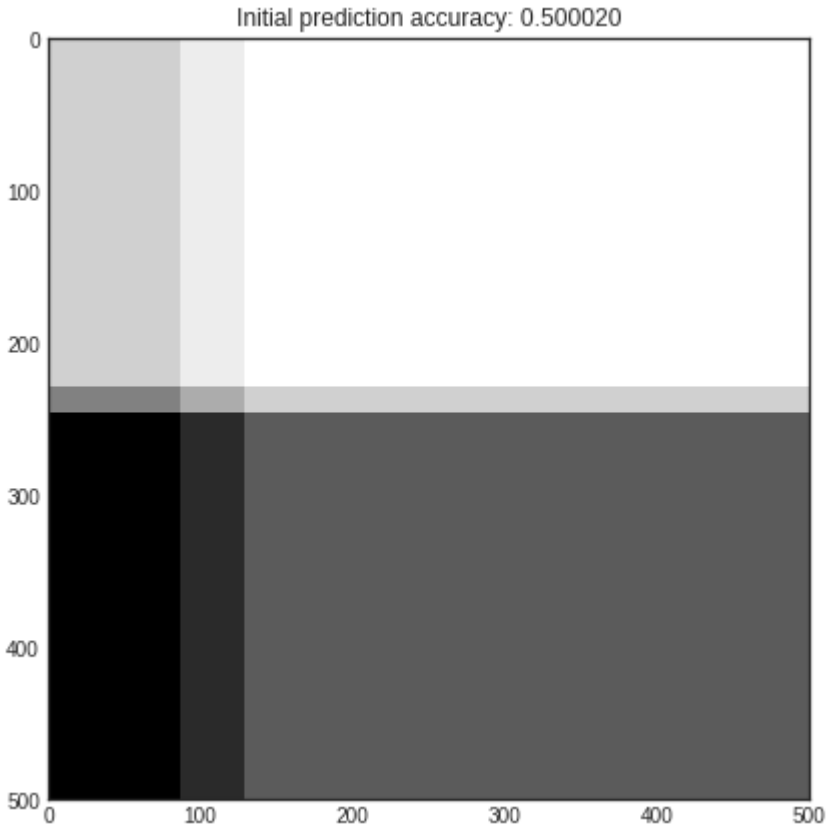
```
# assembling initial training set
n_initial = 5
initial_idx = np.random.choice(range(len(X_full)), size=n_initial, replace=False)
X_train, y_train = X_full[initial_idx], y_full[initial_idx]

# initialize the learner
learner = ActiveLearner(
    estimator=RandomForestClassifier(),
    X_training=X_train, y_training=y_train
)
unqueried_score = learner.score(X_full, y_full)

print('Initial prediction accuracy: %f' % unqueried_score)
```

```
Initial prediction accuracy: 0.500020
```

```
/home/namazu/.local/lib/python3.6/site-packages/sklearn/ensemble/weight_boosting.py:
↪29: DeprecationWarning: numpy.core.umath_tests is an internal NumPy module and␣
↪should not be imported. It will be removed in a future NumPy release.
  from numpy.core.umath_tests import inner1d
```

Let's see how our classifier performs on the initial training set! This is how the class prediction probabilities look like for each pixel.

[4]:
```
# visualizing initial prediciton
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    prediction = learner.predict_proba(X_full)[:, 0]
    plt.imshow(prediction.reshape(im_width, im_height))
    plt.title('Initial prediction accuracy: %f' % unqueried_score)
    plt.show()
```



Now we are going to randomly sample pixels from the image. If the prediction of the pixel's value is uncertain, we query the true value and teach it to the classifier. We are going to do this until we reach at least 90% accuracy.

[5]:
```
from modAL.uncertainty import classifier_uncertainty

performance_history = [unqueried_score]

# learning until the accuracy reaches a given threshold
while learner.score(X_full, y_full) < 0.90:
    stream_idx = np.random.choice(range(len(X_full)))
    if classifier_uncertainty(learner, X_full[stream_idx].reshape(1, -1)) >= 0.4:
        learner.teach(X_full[stream_idx].reshape(1, -1), y_full[stream_idx].reshape(-
↪1, ))
        new_score = learner.score(X_full, y_full)
        performance_history.append(new_score)
        print('Pixel no. %d queried, new accuracy: %f' % (stream_idx, new_score))
```

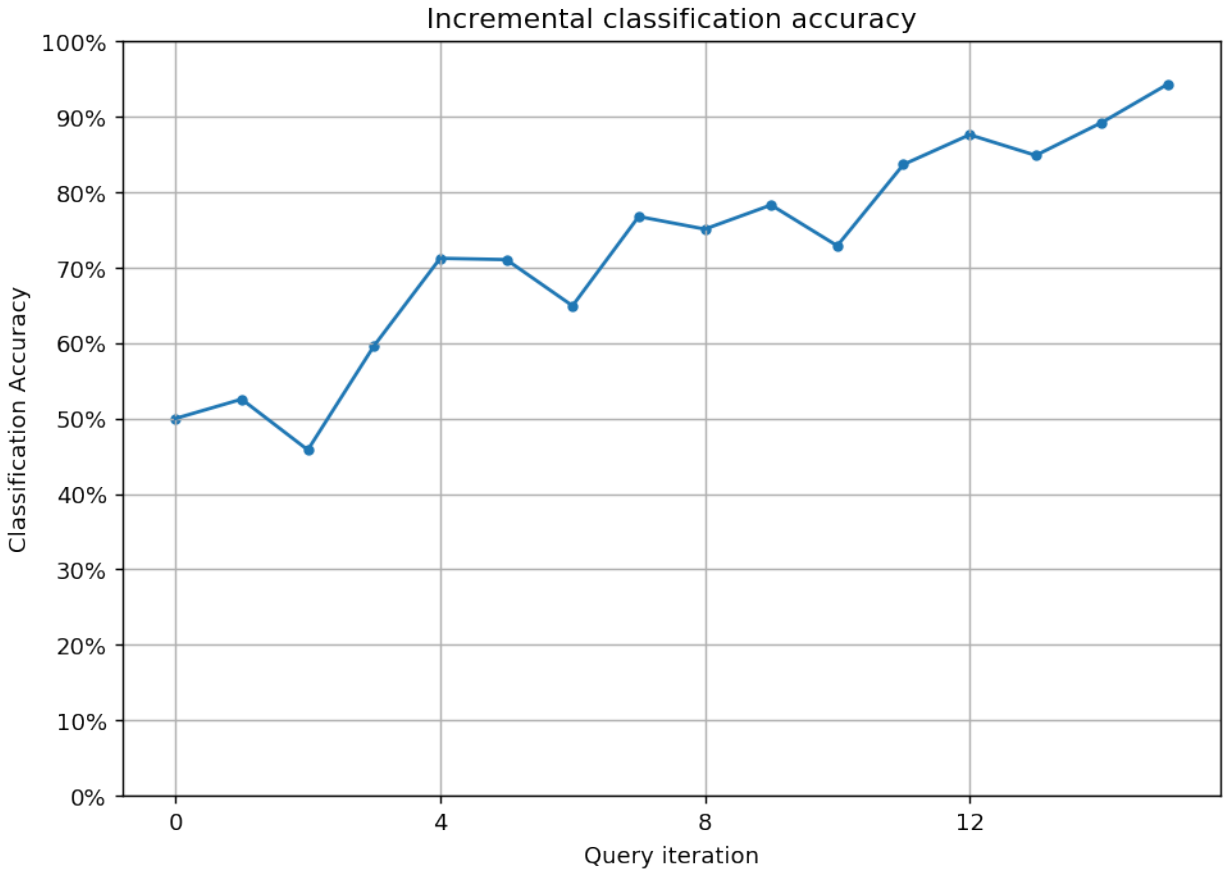
```
Pixel no. 192607 queried, new accuracy: 0.525604
Pixel no. 249064 queried, new accuracy: 0.458388
Pixel no. 235489 queried, new accuracy: 0.596292
Pixel no. 194087 queried, new accuracy: 0.712372
Pixel no. 91580 queried, new accuracy: 0.710612
Pixel no. 199404 queried, new accuracy: 0.649388
Pixel no. 11005 queried, new accuracy: 0.767728
Pixel no. 151961 queried, new accuracy: 0.751044
Pixel no. 17891 queried, new accuracy: 0.783008
Pixel no. 117487 queried, new accuracy: 0.728964
Pixel no. 206353 queried, new accuracy: 0.836684
```

(continues on next page)

(continued from previous page)

```
Pixel no. 154880 queried, new accuracy: 0.876088
Pixel no. 27770 queried, new accuracy: 0.848776
Pixel no. 194658 queried, new accuracy: 0.892492
Pixel no. 178776 queried, new accuracy: 0.943476
```

[6]:
```
# Plot our performance over time.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)

ax.plot(performance_history)
ax.scatter(range(len(performance_history)), performance_history, s=13)

ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=5, integer=True))
ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=10))
ax.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(xmax=1))

ax.set_ylim(bottom=0, top=1)
ax.grid(True)

ax.set_title('Incremental classification accuracy')
ax.set_xlabel('Query iteration')
ax.set_ylabel('Classification Accuracy')

plt.show()
```



CHAPTER 18

Active regression

In this example, we are going to demonstrate how can the ActiveLearner be used for active regression using Gaussian processes. Since Gaussian processes provide a way to quantify uncertainty of the predictions as the covariance function of the process, they can be used in an active learning setting.

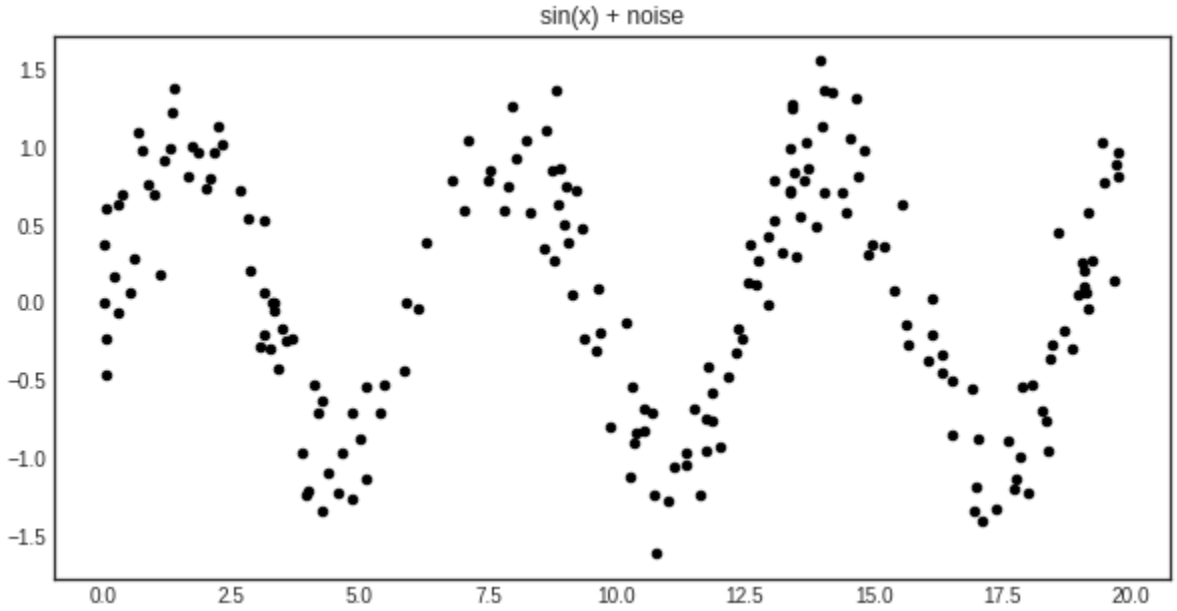
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import WhiteKernel, RBF
from modAL.models import ActiveLearner

%matplotlib inline
```

18.1 The dataset

For this example, we shall try to learn the *noisy sine* function:

[2]:
```
X = np.random.choice(np.linspace(0, 20, 10000), size=200, replace=False).reshape(-1,␣
↪1)
y = np.sin(X) + np.random.normal(scale=0.3, size=X.shape)
```

[3]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 5))
    plt.scatter(X, y, c='k', s=20)
    plt.title('sin(x) + noise')
    plt.show()
```



18.2 Uncertainty measure and query strategy for Gaussian processes

For active learning, we shall define a custom query strategy tailored to Gaussian processes. More information on how to write your custom query strategies can be found at the page Extending modAL. In a nutshell, a *query stategy* in modAL is a function taking (at least) two arguments (an estimator object and a pool of examples), outputting the index of the queried instance and the instance itself. In our case, the arguments are `regressor` and `X`.

[4]:
```
def GP_regression_std(regressor, X):
    _, std = regressor.predict(X, return_std=True)
    query_idx = np.argmax(std)
    return query_idx, X[query_idx]
```

18.3 Active learning

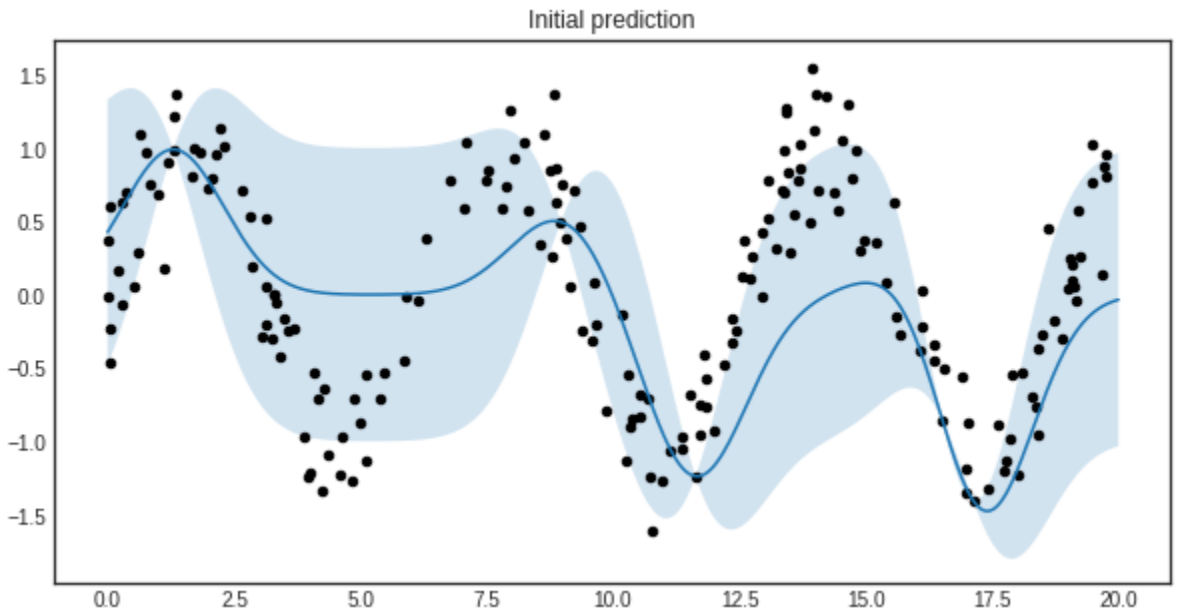
Initializing the active learner is as simple as always.

[5]:
```
n_initial = 5
initial_idx = np.random.choice(range(len(X)), size=n_initial, replace=False)
X_training, y_training = X[initial_idx], y[initial_idx]

kernel = RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e3)) \
         + WhiteKernel(noise_level=1, noise_level_bounds=(1e-10, 1e+1))

regressor = ActiveLearner(
    estimator=GaussianProcessRegressor(kernel=kernel),
    query_strategy=GP_regression_std,
    X_training=X_training.reshape(-1, 1), y_training=y_training.reshape(-1, 1)
)
```

The initial regressor is not very accurate.

[6]:
```
X_grid = np.linspace(0, 20, 1000)
y_pred, y_std = regressor.predict(X_grid.reshape(-1, 1), return_std=True)
y_pred, y_std = y_pred.ravel(), y_std.ravel()
```

[7]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 5))
    plt.plot(X_grid, y_pred)
    plt.fill_between(X_grid, y_pred - y_std, y_pred + y_std, alpha=0.2)
    plt.scatter(X, y, c='k', s=20)
    plt.title('Initial prediction')
    plt.show()
```

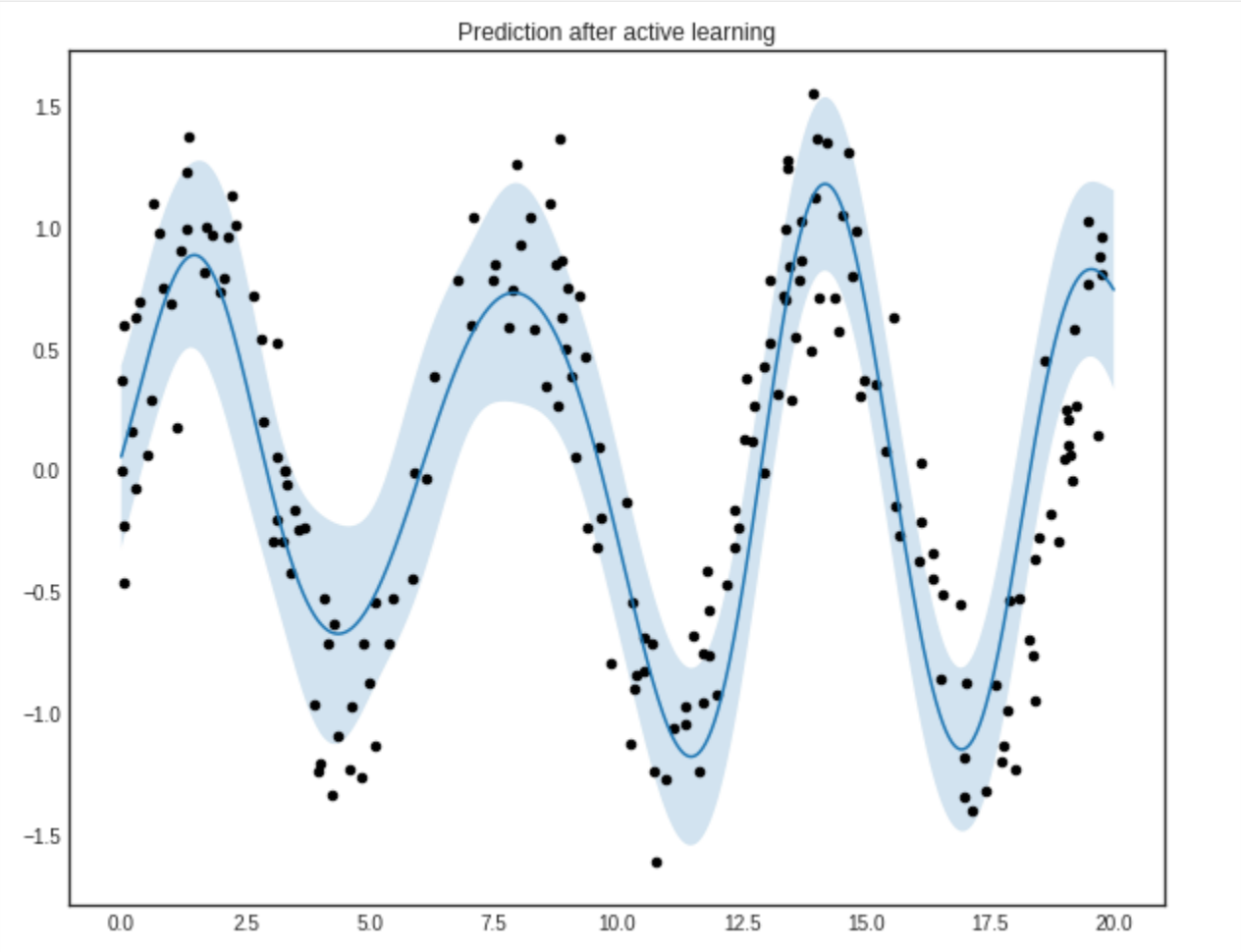


The blue band enveloping the regressor represents the standard deviation of the Gaussian process at the given point. Now we are ready to do active learning!

[8]:
```
n_queries = 10
for idx in range(n_queries):
    query_idx, query_instance = regressor.query(X)
    regressor.teach(X[query_idx].reshape(1, -1), y[query_idx].reshape(1, -1))
```

[9]:
```
y_pred_final, y_std_final = regressor.predict(X_grid.reshape(-1, 1), return_std=True)
y_pred_final, y_std_final = y_pred_final.ravel(), y_std_final.ravel()
```

[10]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 8))
    plt.plot(X_grid, y_pred_final)
    plt.fill_between(X_grid, y_pred_final - y_std_final, y_pred_final + y_std_final,␣
↪alpha=0.2)
    plt.scatter(X, y, c='k', s=20)
    plt.title('Prediction after active learning')
    plt.show()
```


Prediction after active learning
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

CHAPTER 19

Ensemble regression

With an ensemble of regressors, the standard deviation of the predictions at a given point can be thought of as a measure of disagreement. This can be used for active regression. In the following example, we are going to see how can it be done using the CommitteeRegressor class.

The executable script for this example can be found here!

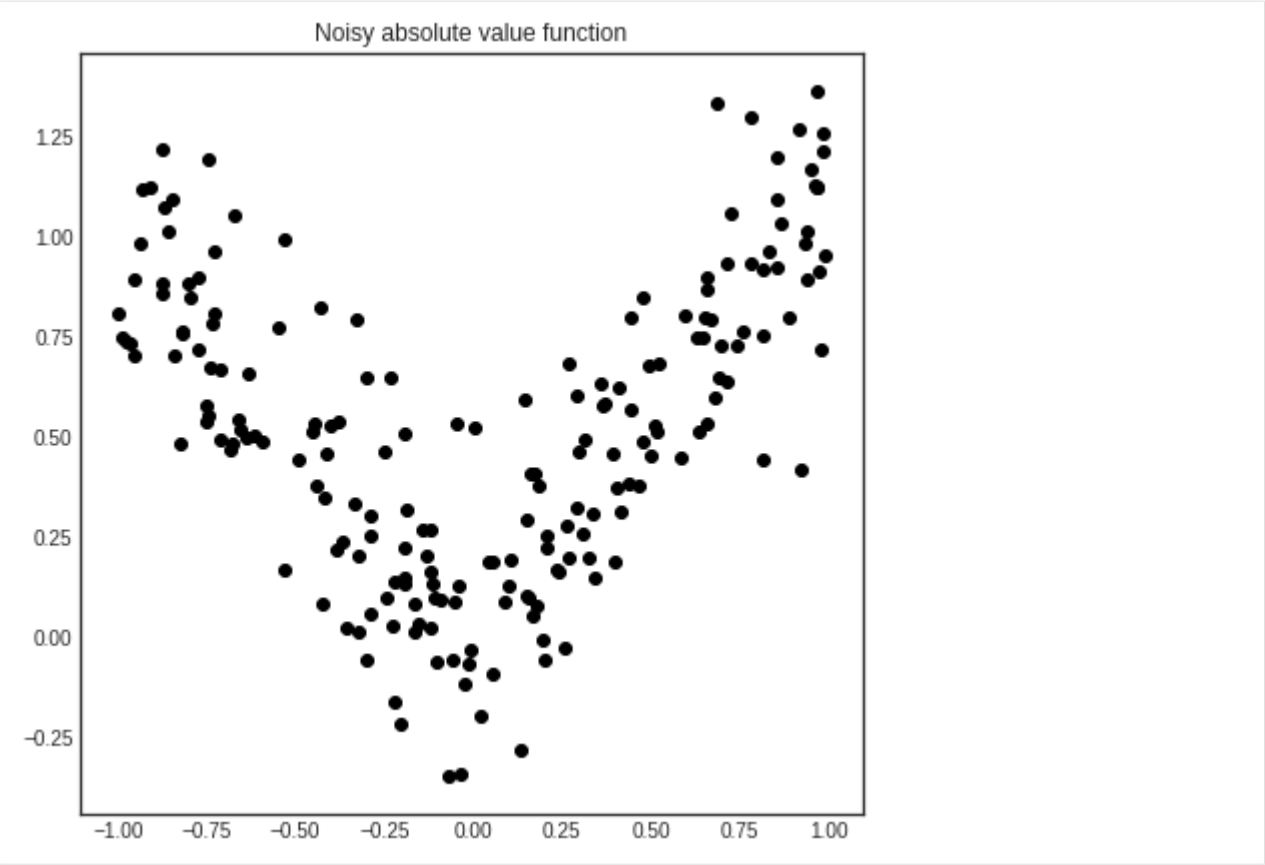
19.1 The dataset

For the sake of this example, we are going to learn the *noisy absolute value* function.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import WhiteKernel, RBF
from modAL.models import ActiveLearner, CommitteeRegressor
from modAL.disagreement import max_std_sampling
```

[2]:
```
# generating the data
X = np.concatenate((np.random.rand(100)-1, np.random.rand(100)))
y = np.abs(X) + np.random.normal(scale=0.2, size=X.shape)
```

[3]:
```
# visualizing the data
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    plt.scatter(X, y, c='k')
    plt.title('Noisy absolute value function')
    plt.show()
```



19.2 Measuring disagreement with CommitteeRegression

If you have several regressors, measuring disagreement can be done by calculating the standard deviation of the predictions for each point. This of course cannot be achieved with classifier algorithms, where averaging the class labels doesn't make sense. (Or it is undefined even, if the class labels are strings for example.) In the simplest setting, this is implemented in the function `modAL.disagreement.max_std_sampling`.

This measure is default for CommitteeRegressors, so we don't need to specify this upon initialization.

19.3 Active regression

With an ensemble of regressors, it can happen that each one explains part of your data particularly well, while doing poorly on the rest. In our case, it can happen when one of them only seen negative numbers and the other only seen positive ones.

[4]:
```
# initializing the regressors
n_initial = 10
kernel = RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e3)) \
         + WhiteKernel(noise_level=1, noise_level_bounds=(1e-10, 1e+1))

initial_idx = list()
initial_idx.append(np.random.choice(range(100), size=n_initial, replace=False))
initial_idx.append(np.random.choice(range(100, 200), size=n_initial, replace=False))
```

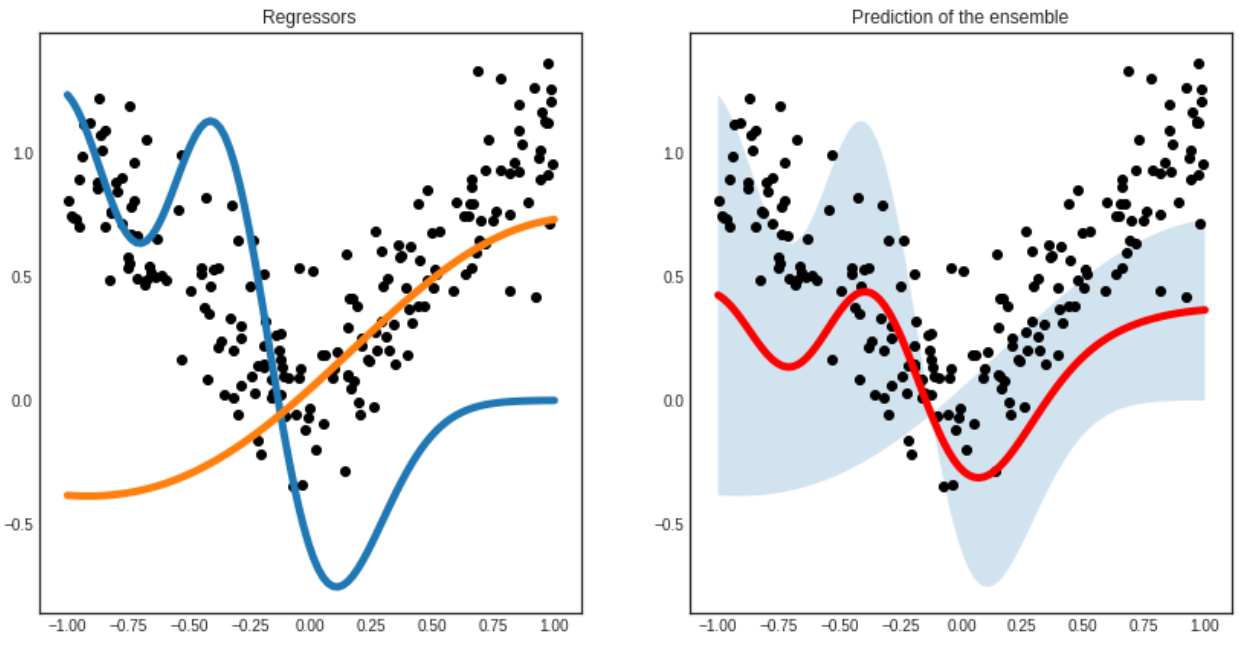
(continues on next page)

(continued from previous page)

```
learner_list = [ActiveLearner(
                        estimator=GaussianProcessRegressor(kernel),
                        X_training=X[idx].reshape(-1, 1), y_training=y[idx].reshape(-
↪1, 1)
                    )
                    for idx in initial_idx]
```

[5]:
```
# initializing the Committee
committee = CommitteeRegressor(
    learner_list=learner_list,
    query_strategy=max_std_sampling
)
```

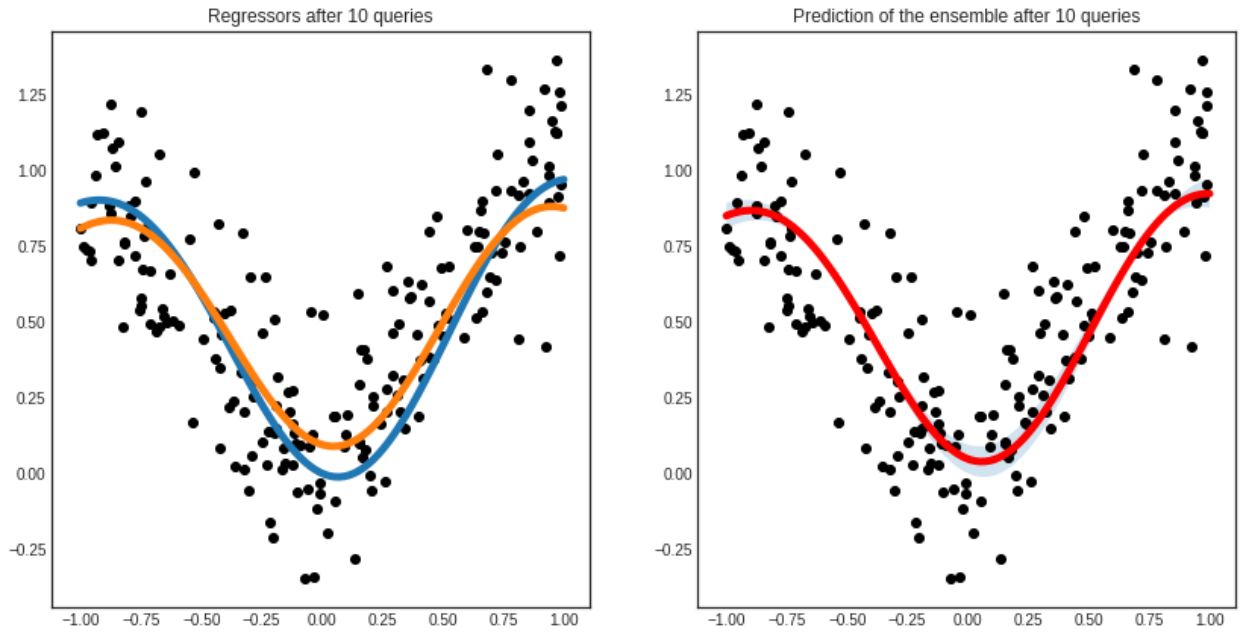
[6]:
```
# visualizing the regressors
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(14, 7))
    x = np.linspace(-1, 1, 100)

    plt.subplot(1, 2, 1)
    for learner_idx, learner in enumerate(committee):
        plt.plot(x, learner.predict(x.reshape(-1, 1)), linewidth=5)
    plt.scatter(X, y, c='k')
    plt.title('Regressors')

    plt.subplot(1, 2, 2)
    pred, std = committee.predict(x.reshape(-1, 1), return_std=True)
    pred = pred.reshape(-1, )
    std = std.reshape(-1, )
    plt.plot(x, pred, c='r', linewidth=5)
    plt.fill_between(x, pred - std, pred + std, alpha=0.2)
    plt.scatter(X, y, c='k')
    plt.title('Prediction of the ensemble')
    plt.show()
```



After a few queries, the differences are smoothed out and the regressors reach an agreement.

[7]:
```
# active regression
n_queries = 10
for idx in range(n_queries):
    query_idx, query_instance = committee.query(X.reshape(-1, 1))
    committee.teach(X[query_idx].reshape(-1, 1), y[query_idx].reshape(-1, 1))
```

[8]:
```
# visualizing the regressors
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(14, 7))
    x = np.linspace(-1, 1, 100)

    plt.subplot(1, 2, 1)
    for learner_idx, learner in enumerate(committee):
        plt.plot(x, learner.predict(x.reshape(-1, 1)), linewidth=5)
    plt.scatter(X, y, c='k')
    plt.title('Regressors after %d queries' % n_queries)

    plt.subplot(1, 2, 2)
    pred, std = committee.predict(x.reshape(-1, 1), return_std=True)
    pred = pred.reshape(-1, )
    std = std.reshape(-1, )
    plt.plot(x, pred, c='r', linewidth=5)
    plt.fill_between(x, pred - std, pred + std, alpha=0.2)
    plt.scatter(X, y, c='k')
    plt.title('Prediction of the ensemble after %d queries' % n_queries)
    plt.show()
```



[ ]:

CHAPTER 20

Bayesian optimization

When a function is expensive to evaluate, or when gradients are not available, optimalizing it requires more sophisticated methods than gradient descent. One such method is Bayesian optimization, which lies close to active learning. In Bayesian optimization, instead of picking queries by maximizing the uncertainty of predictions, function values are evaluated at points where the promise of finding a better value is large. In modAL, these algorithms are implemented with the `BayesianOptimizer` class, which is a sibling of `ActiveLearner`. In the following example, their use is demonstrated on a toy problem.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from functools import partial
from sklearn.gaussian_process import GaussianProcessRegressor
from sklearn.gaussian_process.kernels import Matern
from modAL.models import BayesianOptimizer
from modAL.acquisition import optimizer_EI, max_EI

%matplotlib inline
```

20.1 The function to be optimized

We are going to optimize a simple function to demonstrate the use of `BayesianOptimizer`.

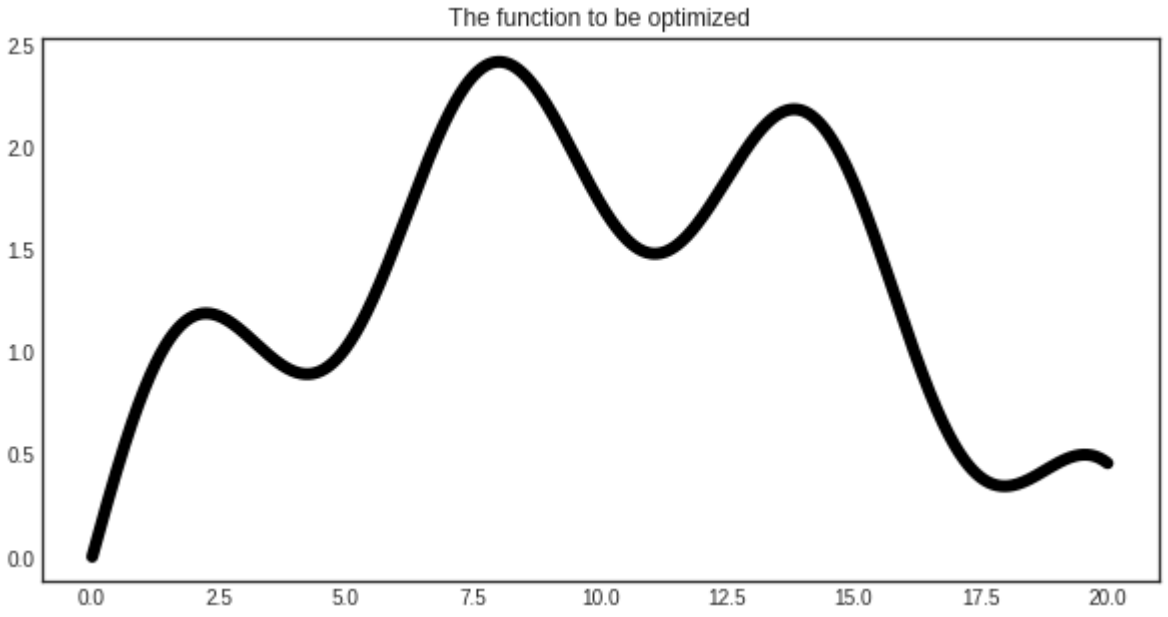
[2]:
```
import numpy as np

# generating the data
X = np.linspace(0, 20, 1000).reshape(-1, 1)
y = np.sin(X)/2 - ((10 - X)**2)/50 + 2
```

[3]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 5))
    plt.plot(X, y, c='k', linewidth=6)
```

(continues on next page)

(continued from previous page)

```
    plt.title('The function to be optimized')
    plt.show()
```



20.2 Gaussian processes

In Bayesian optimization, usually a Gaussian process regressor is used to predict the function to be optimized. One reason is that Gaussian processes can estimate the uncertainty of the prediction at a given point. This in turn can be used to estimate the possible gains at the unknown points.

[4]:
```
# assembling initial training set
X_initial, y_initial = X[150].reshape(1, -1), y[150].reshape(1, -1)

# defining the kernel for the Gaussian process
kernel = Matern(length_scale=1.0)
regressor = GaussianProcessRegressor(kernel=kernel)
```

20.3 Optimizing using *expected improvement*

During the optimization, the utility of each point is given by the so-called *acquisition function*. In this case, we are going to use the *expected improvement*, which is defined by

$$EI(x) = (\mu(x) - f(x^+))\psi\Big(\frac{\mu(x) - f(x^+)}{\sigma(x)}\Big) + \sigma(x)\phi\Big(\frac{\mu(x) - f(x^+)}{\sigma(x)}\Big),$$

where $\mu(x)$ and $\sigma(x)$ are the mean and variance of the Gaussian process regressor at x, f is the function to be optimized with estimated maximum at x^+, and $\psi(z)$, $\phi(z)$ denotes the cumulative distribution function and density function of a standard Gaussian distribution. After each query, the acquisition function is reevaluated and the new query is chosen to maximize the acquisition function.

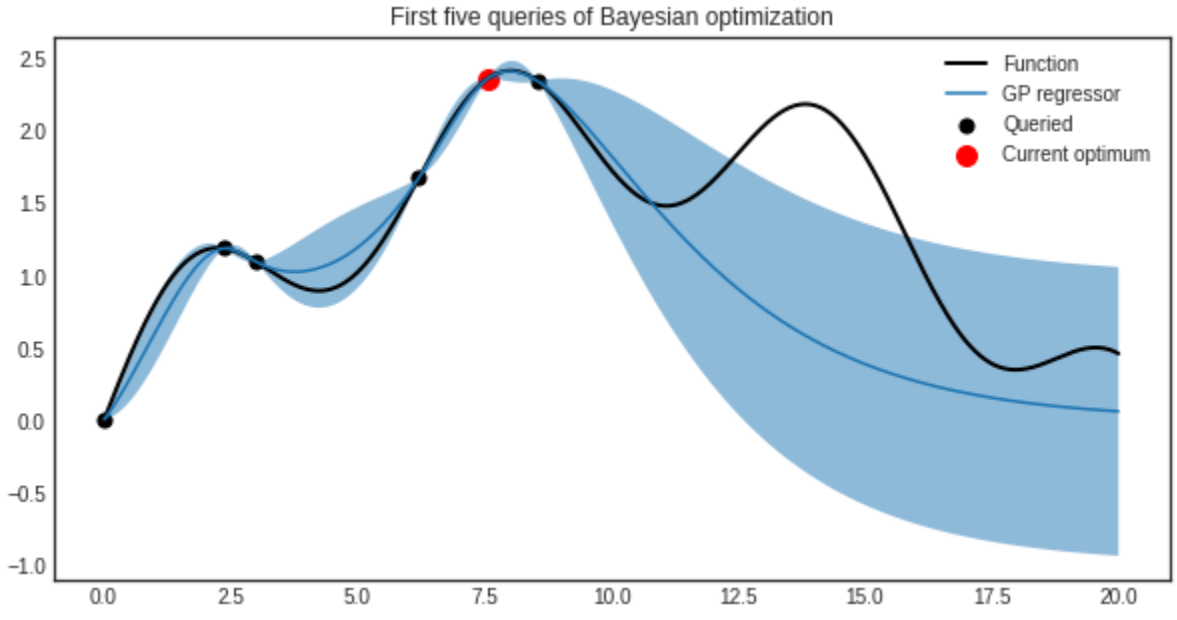
[5]:
```
# initializing the optimizer
optimizer = BayesianOptimizer(
    estimator=regressor,
    X_training=X_initial, y_training=y_initial,
    query_strategy=max_EI
)
```

[6]:
```
# Bayesian optimization
for n_query in range(5):
    query_idx, query_inst = optimizer.query(X)
    optimizer.teach(X[query_idx].reshape(1, -1), y[query_idx].reshape(1, -1))
```

Using *expected improvement*, the first five queries are the following.

[7]:
```
y_pred, y_std = optimizer.predict(X, return_std=True)
y_pred, y_std = y_pred.ravel(), y_std.ravel()
X_max, y_max = optimizer.get_max()
```

[8]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(10, 5))
    plt.scatter(optimizer.X_training, optimizer.y_training, c='k', s=50, label=
↪'Queried')
    plt.scatter(X_max, y_max, s=100, c='r', label='Current optimum')
    plt.plot(X.ravel(), y, c='k', linewidth=2, label='Function')
    plt.plot(X.ravel(), y_pred, label='GP regressor')
    plt.fill_between(X.ravel(), y_pred - y_std, y_pred + y_std, alpha=0.5)
    plt.title('First five queries of Bayesian optimization')
    plt.legend()
    plt.show()
```



20.4 Acquisition functions

Currently, there are three built in acquisition functions in the `modAL.acquisition` module: *expected improvement*, *probability of improvement* and *upper confidence bounds*. You can find them in detail here.

CHAPTER 21

Query by committee

Query by committee is another popular active learning strategy, which alleviates many disadvantages of uncertainty sampling. For instance, uncertainty sampling tends to be biased towards the actual learner and it may miss important examples which are not in the sight of the estimator. This is fixed by keeping several hypotheses at the same time, selecting queries where disagreement occurs between them. In this example, we shall see how this works in the simplest case, using the iris dataset.

The executable script for this example is available here!

To enforce a reproducible result across runs, we set a random seed.

[1]:
```
import numpy as np

# Set our RNG seed for reproducibility.
RANDOM_STATE_SEED = 1
np.random.seed(RANDOM_STATE_SEED)
```

21.1 The dataset

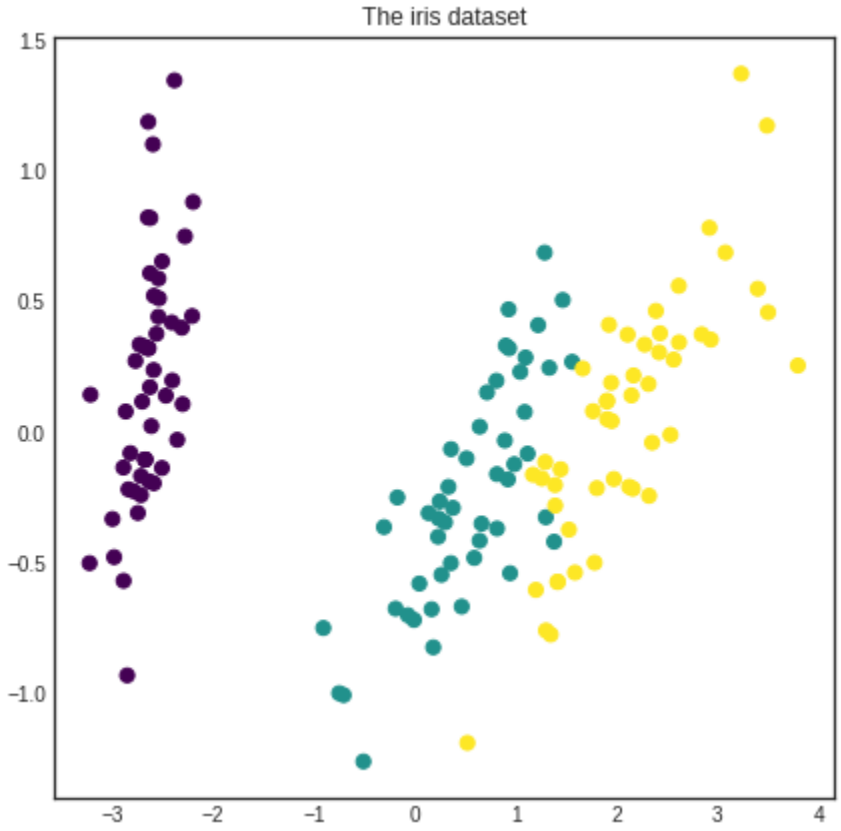
We are going to use the iris dataset for this example. For more information on the iris dataset, see its wikipedia page. For its scikit-learn interface, see the scikit-learn documentation.

[2]:
```
import matplotlib as mpl
import matplotlib.pyplot as plt
%matplotlib inline
from sklearn.decomposition import PCA
from sklearn.datasets import load_iris

# loading the iris dataset
iris = load_iris()

# visualizing the classes
with plt.style.context('seaborn-white'):
```

(continues on next page)

(continued from previous page)

```
    plt.figure(figsize=(7, 7))
    pca = PCA(n_components=2).fit_transform(iris['data'])
    plt.scatter(x=pca[:, 0], y=pca[:, 1], c=iris['target'], cmap='viridis', s=50)
    plt.title('The iris dataset')
    plt.show()
```



21.2 Initializing the Committee

In this example, we are going to use the `Committee` class from `modAL.models`. Its interface is almost exactly identical to the `ActiveLearner`. Upon initialization, `Committee` requires a list of active learners. First, we generate the pool of unlabeled data.

[3]:
```
from copy import deepcopy

# generate the pool
X_pool = deepcopy(iris['data'])
y_pool = deepcopy(iris['target'])
```

Now we are ready to initialize the `Committee`. For this, we need a list of `ActiveLearner` objects, which we will define now.

[4]:
```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
```

(continues on next page)

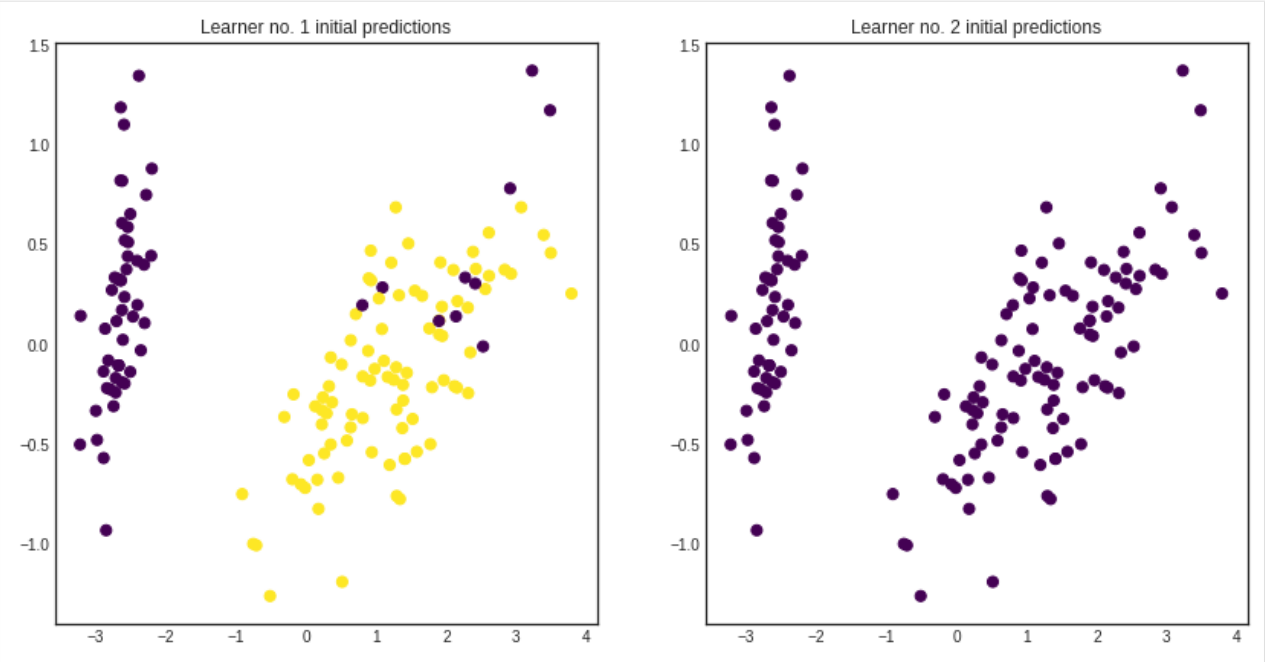
(continued from previous page)

```
from modAL.models import ActiveLearner, Committee

# initializing Committee members
n_members = 2
learner_list = list()

for member_idx in range(n_members):
    # initial training data
    n_initial = 2
    train_idx = np.random.choice(range(X_pool.shape[0]), size=n_initial,␣
↪replace=False)
    X_train = X_pool[train_idx]
    y_train = y_pool[train_idx]

    # creating a reduced copy of the data with the known instances removed
    X_pool = np.delete(X_pool, train_idx, axis=0)
    y_pool = np.delete(y_pool, train_idx)

    # initializing learner
    learner = ActiveLearner(
        estimator=RandomForestClassifier(),
        X_training=X_train, y_training=y_train
    )
    learner_list.append(learner)

# assembling the committee
committee = Committee(learner_list=learner_list)
```

```
/home/namazu/.local/lib/python3.6/site-packages/sklearn/ensemble/weight_boosting.py:
↪29: DeprecationWarning: numpy.core.umath_tests is an internal NumPy module and␣
↪should not be imported. It will be removed in a future NumPy release.
  from numpy.core.umath_tests import inner1d
```

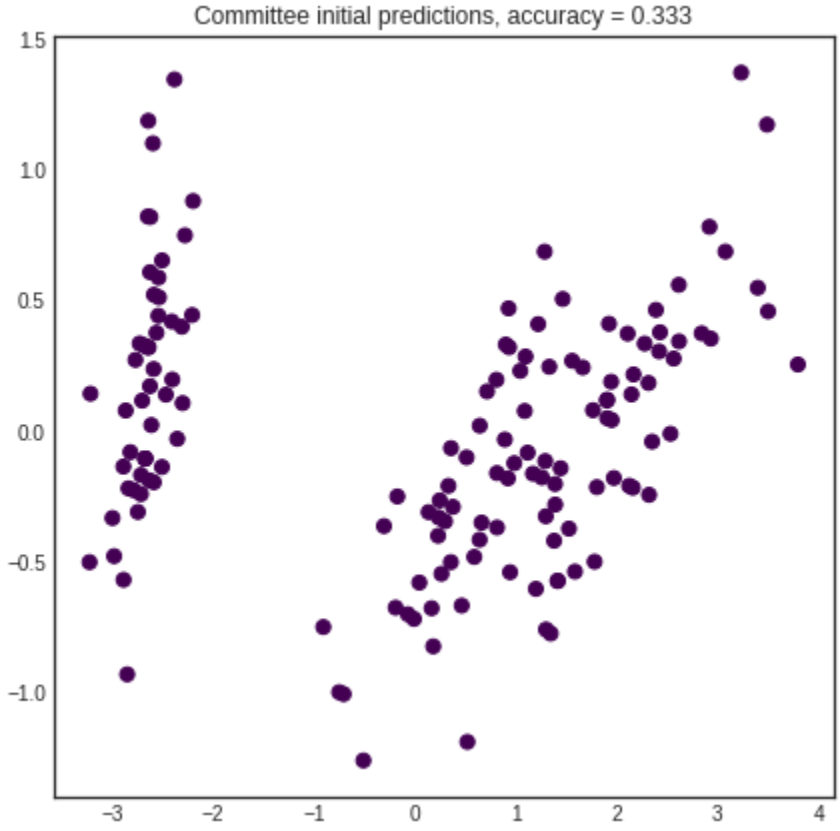
As you can see, the various hypotheses (which are taking the form of ActiveLearners) can be quite different.

[5]:
```
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(n_members*7, 7))
    for learner_idx, learner in enumerate(committee):
        plt.subplot(1, n_members, learner_idx + 1)
        plt.scatter(x=pca[:, 0], y=pca[:, 1], c=learner.predict(iris['data']), cmap=
↪'viridis', s=50)
        plt.title('Learner no. %d initial predictions' % (learner_idx + 1))
    plt.show()
```



Prediction is done by averaging the class probabilities for each learner and chosing the most likely class.

[6]:
```
unqueried_score = committee.score(iris['data'], iris['target'])

with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    prediction = committee.predict(iris['data'])
    plt.scatter(x=pca[:, 0], y=pca[:, 1], c=prediction, cmap='viridis', s=50)
    plt.title('Committee initial predictions, accuracy = %1.3f' % unqueried_score)
    plt.show()
```



21.3 Active learning

The active learning loop is the same as for the `ActiveLearner`.

[7]:
```
performance_history = [unqueried_score]

# query by committee
n_queries = 20
for idx in range(n_queries):
    query_idx, query_instance = committee.query(X_pool)
    committee.teach(
        X=X_pool[query_idx].reshape(1, -1),
        y=y_pool[query_idx].reshape(1, )
    )
    performance_history.append(committee.score(iris['data'], iris['target']))
    # remove queried instance from pool
    X_pool = np.delete(X_pool, query_idx, axis=0)
    y_pool = np.delete(y_pool, query_idx)
```

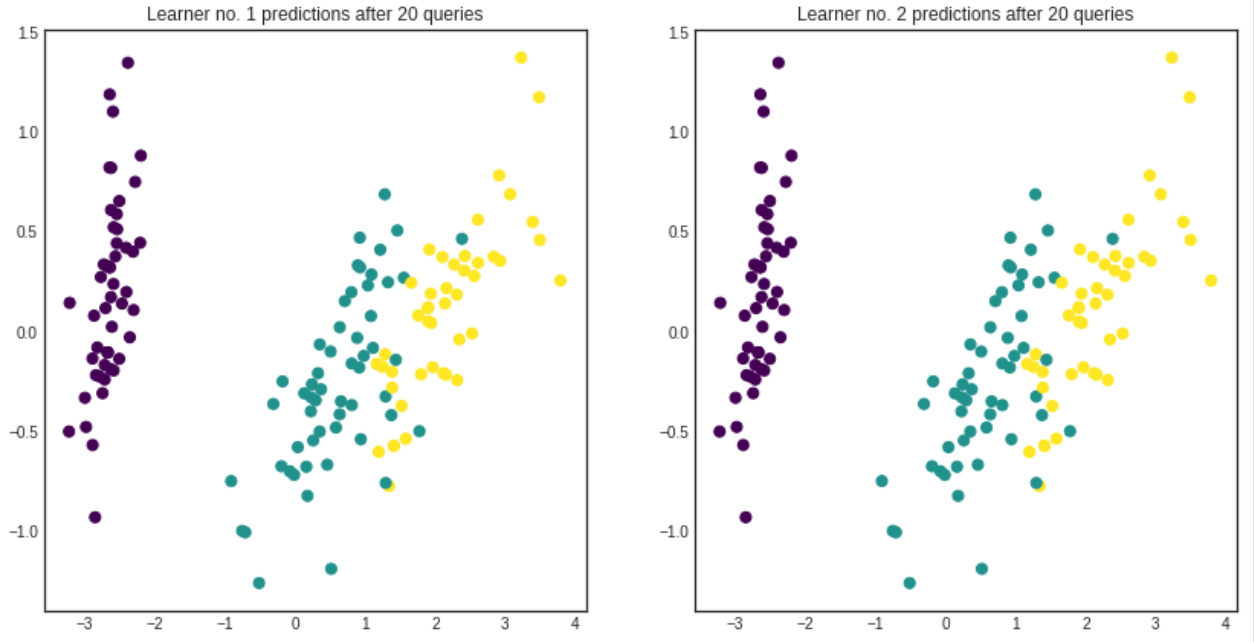
After a few queries, the hypotheses straighten out their disagreements and they reach consensus. Prediction accuracy is greatly improved in this case.

[8]:
```
# visualizing the final predictions per learner
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(n_members*7, 7))
```

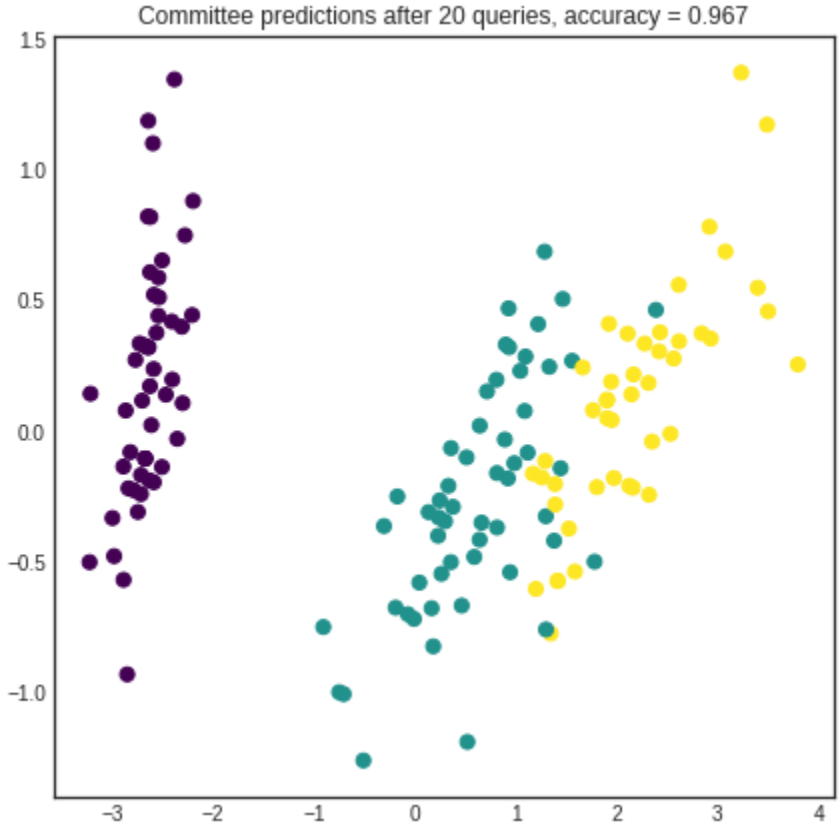
(continues on next page)

(continued from previous page)

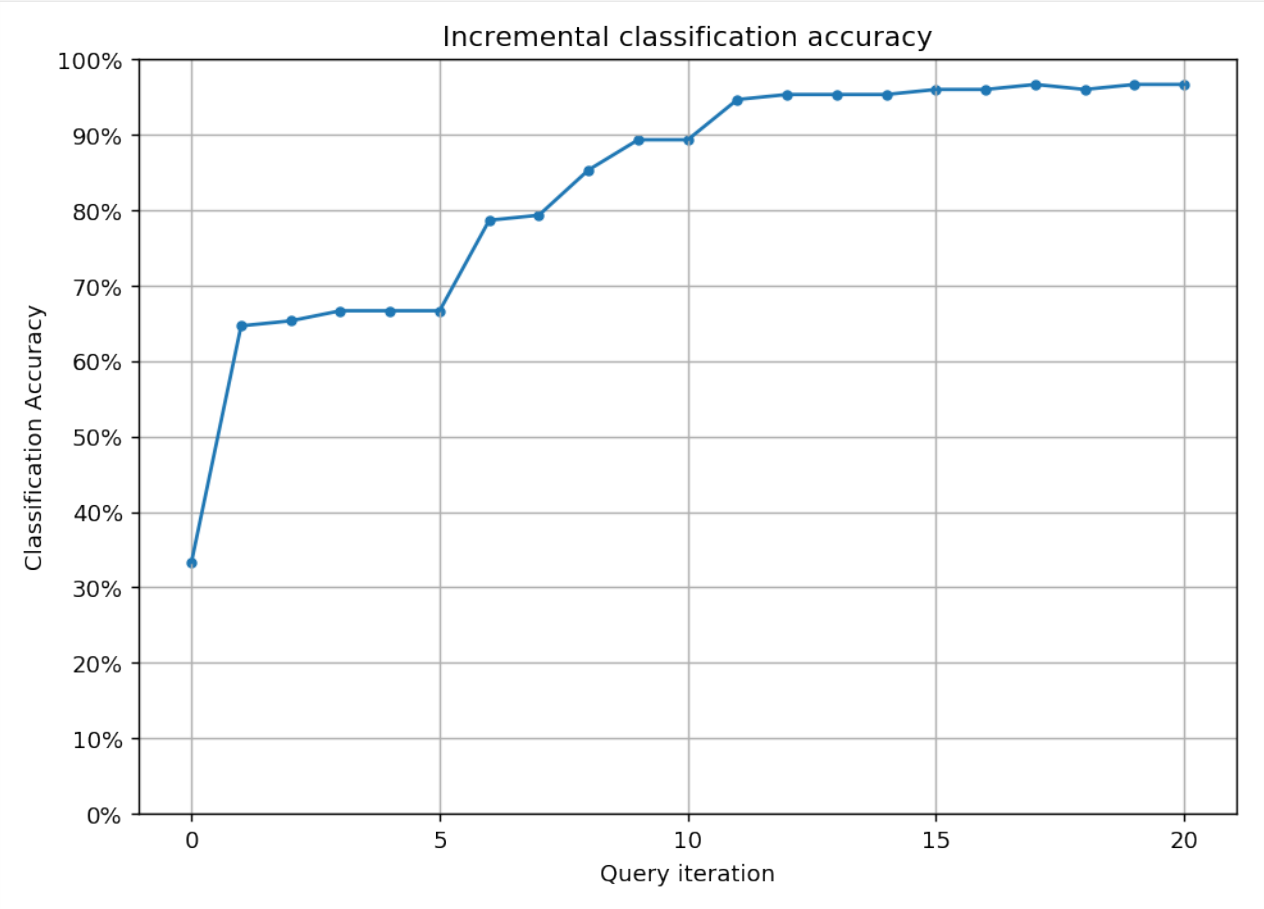
```
    for learner_idx, learner in enumerate(committee):
        plt.subplot(1, n_members, learner_idx + 1)
        plt.scatter(x=pca[:, 0], y=pca[:, 1], c=learner.predict(iris['data']), cmap=
↪'viridis', s=50)
        plt.title('Learner no. %d predictions after %d queries' % (learner_idx + 1, n_
↪queries))
    plt.show()
```



[9]:
```
# visualizing the Committee's predictions
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    prediction = committee.predict(iris['data'])
    plt.scatter(x=pca[:, 0], y=pca[:, 1], c=prediction, cmap='viridis', s=50)
    plt.title('Committee predictions after %d queries, accuracy = %1.3f'
              % (n_queries, committee.score(iris['data'], iris['target'])))
    plt.show()
```



[10]:
```
# Plot our performance over time.
fig, ax = plt.subplots(figsize=(8.5, 6), dpi=130)

ax.plot(performance_history)
ax.scatter(range(len(performance_history)), performance_history, s=13)

ax.xaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=5, integer=True))
ax.yaxis.set_major_locator(mpl.ticker.MaxNLocator(nbins=10))
ax.yaxis.set_major_formatter(mpl.ticker.PercentFormatter(xmax=1))

ax.set_ylim(bottom=0, top=1)
ax.grid(True)

ax.set_title('Incremental classification accuracy')
ax.set_xlabel('Query iteration')
ax.set_ylabel('Classification Accuracy')

plt.show()
```


Incremental classification accuracy
Classification Accuracy
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
0
5
10
15
20
Query iteration

CHAPTER 22

Bootstrapping and bagging

Bootstrapping and bagging can be very useful when using ensemble models such as the Committee. In essence, bootstrapping is random sampling with replacement from the available training data. Bagging (= bootstrap aggregation) is performing it many times and training an estimator for each bootstrapped dataset. It is available in modAL for both the base ActiveLearner model and the Committee model as well. In this short tutorial, we are going to see how to perform bootstrapping and bagging in your active learning workflow.

The executable script for this example can be found here!

To enforce a reproducible result across runs, we set a random seed.

[1]:
```
import numpy as np

# Set our RNG seed for reproducibility.
RANDOM_STATE_SEED = 123
np.random.seed(RANDOM_STATE_SEED)
```

22.1 The dataset

In this short example, we will try to learn the shape of three black disks on a white background.

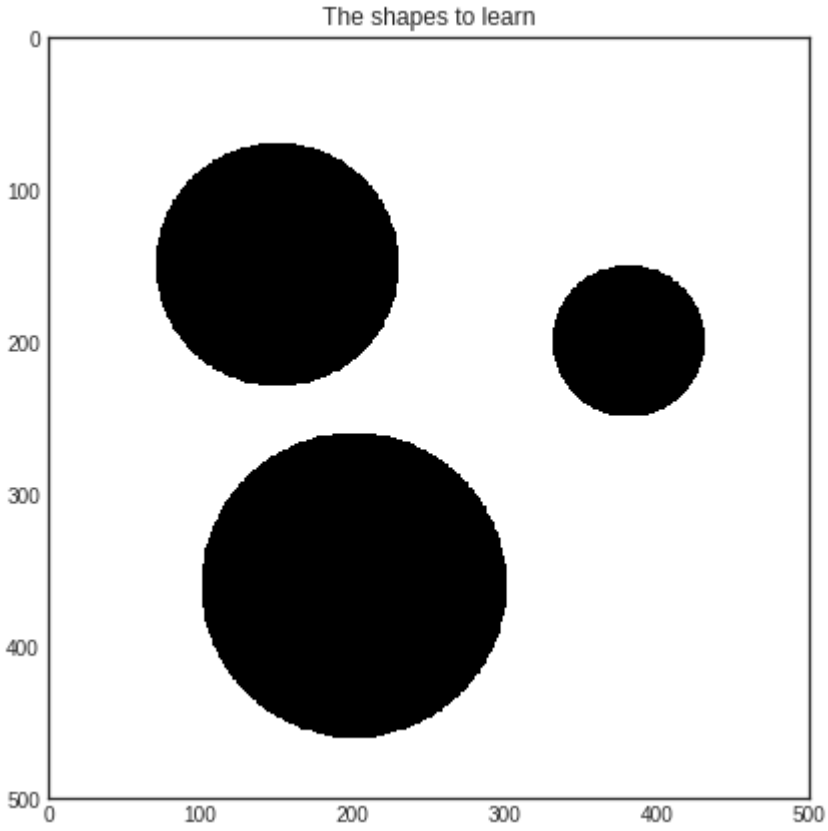
[2]:
```
import numpy as np
from itertools import product

# creating the dataset
im_width = 500
im_height = 500
data = np.zeros((im_height, im_width))
# each disk is coded as a triple (x, y, r), where x and y are the centers and r is␣
↪the radius
disks = [(150, 150, 80), (200, 380, 50), (360, 200, 100)]
for i, j in product(range(im_width), range(im_height)):
    for x, y, r in disks:
```

(continues on next page)

(continued from previous page)

```
        if (x-i)**2 + (y-j)**2 < r**2:
            data[i, j] = 1
```

[3]:
```
import matplotlib.pyplot as plt
%matplotlib inline

# visualizing the dataset
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    plt.imshow(data)
    plt.title('The shapes to learn')
    plt.show()
```



22.2 Initializing the learners with bootstrapping

First we shall train three ActiveLearners on a bootstrapped dataset. Then we are going to bundle them together in a Committee and see how bagging is done with modAL.

[4]:
```
# create the pool from the image
X_pool = np.transpose(
    [np.tile(np.asarray(range(data.shape[0])), data.shape[1]),
     np.repeat(np.asarray(range(data.shape[1])), data.shape[0])]
)
# map the intensity values against the grid
```

(continues on next page)

(continued from previous page)

```
y_pool = np.asarray([data[P[0], P[1]] for P in X_pool])

# initial training data
initial_idx = np.random.choice(range(len(X_pool)), size=500)
```

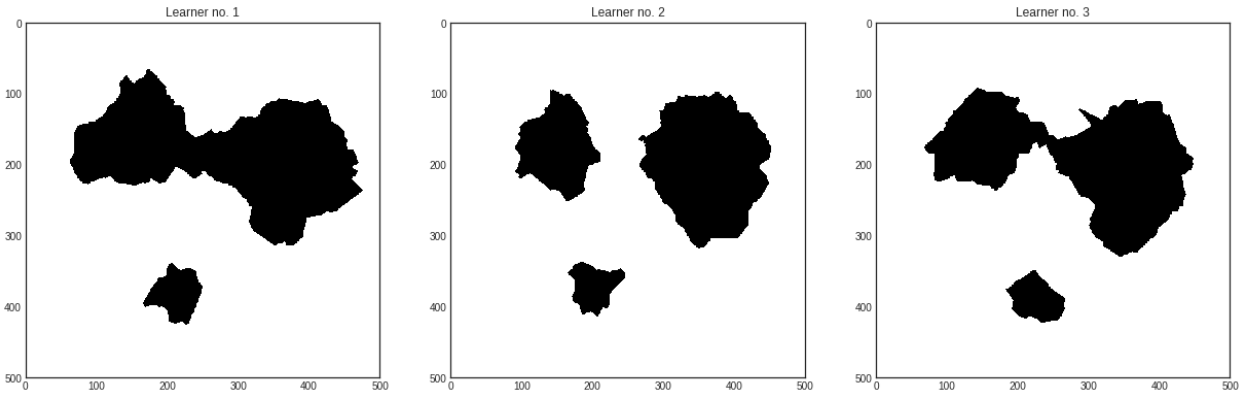
[5]:
```
from sklearn.neighbors import KNeighborsClassifier
from modAL.models import ActiveLearner, Committee

# initializing the learners
n_learners = 3
learner_list = []
for _ in range(n_learners):
    learner = ActiveLearner(
        estimator=KNeighborsClassifier(n_neighbors=10),
        X_training=X_pool[initial_idx], y_training=y_pool[initial_idx],
        bootstrap_init=True
    )
    learner_list.append(learner)
```

As you can see, the main difference in this from the regular use of ActiveLearner is passing `bootstrap_init=True` upon initialization. This makes it to train the model on a bootstrapped dataset, although it stores the *complete* training dataset among its known examples. In this exact case, here is how the classifiers perform:

We can put our learners together in a Committee and see how they perform.

[6]:
```
# assembling the Committee
committee = Committee(learner_list)
```

[7]:
```
# visualizing every learner in the Committee
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7*n_learners, 7))
    for learner_idx, learner in enumerate(committee):
        plt.subplot(1, n_learners, learner_idx+1)
        plt.imshow(learner.predict(X_pool).reshape(im_height, im_width))
        plt.title('Learner no. %d' % (learner_idx + 1))
    plt.show()
```

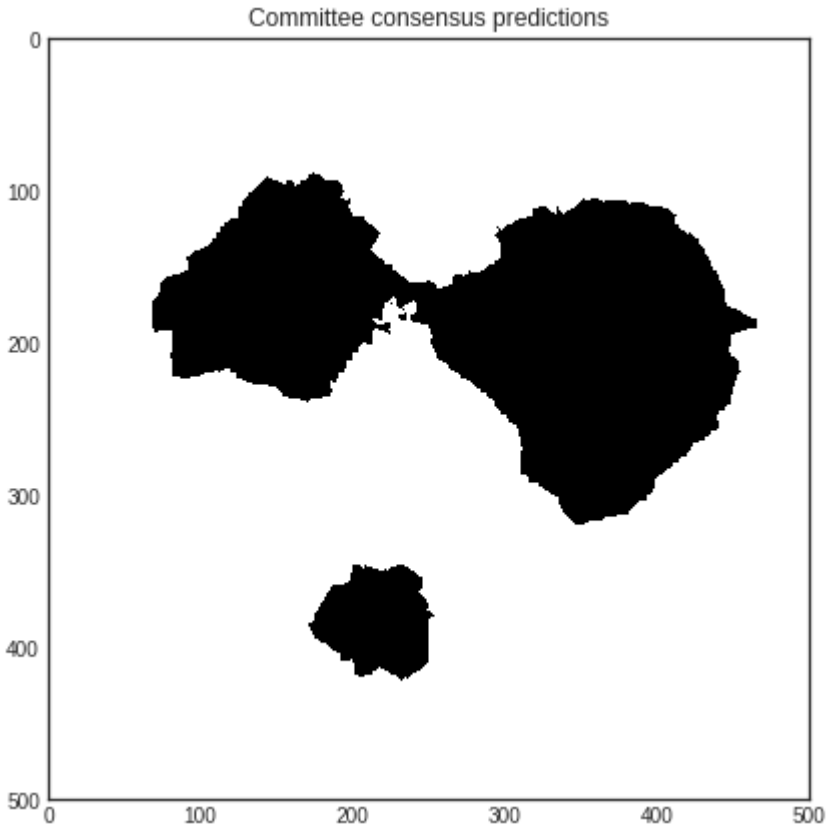


[8]:
```
# visualizing the Committee's predictions
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7, 7))
    plt.imshow(committee.predict(X_pool).reshape(im_height, im_width))
```

(continues on next page)

(continued from previous page)

```
    plt.title('Committee consensus predictions')
    plt.show()
```



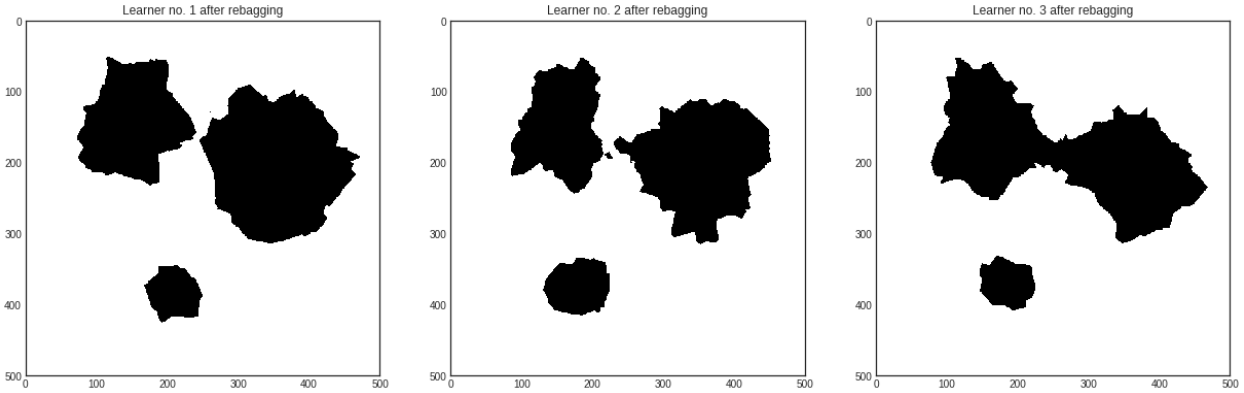
22.3 Bagging

If you would like to take each learner in the Committee and retrain them using bagging, you can use the `.rebag()` method:

[9]:
```
committee.rebag()
```

In this case, the classifiers perform in the following way after rebagging.

[10]:
```
# visualizing the learners in the retrained Committee
with plt.style.context('seaborn-white'):
    plt.figure(figsize=(7*n_learners, 7))
    for learner_idx, learner in enumerate(committee):
        plt.subplot(1, n_learners, learner_idx+1)
        plt.imshow(learner.predict(X_pool).reshape(im_height, im_width))
        plt.title('Learner no. %d after rebagging' % (learner_idx + 1))
    plt.show()
```


Learner no. 1 after rebagging
Learner no. 2 after rebagging
Learner no. 3 after rebagging
0
100
200
300
400
500

[ ]:

CHAPTER 23

Keras models in modAL workflows

Thanks for the scikit-learn API of Keras, you can seamlessly integrate Keras models into your modAL workflow. In this tutorial, we shall quickly introduce how to use the scikit-learn API of Keras and we are going to see how to do active learning with it. More details on the Keras scikit-learn API can be found here.

The executable script for this example can be found here!

23.1 Keras' scikit-learn API

By default, a Keras model's interface differs from what is used for scikit-learn estimators. However, with the use of its scikit-learn wrapper, it is possible to adapt your model.

[1]:
```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D, MaxPooling2D
from keras.wrappers.scikit_learn import KerasClassifier

# build function for the Keras' scikit-learn API
def create_keras_model():
    """
    This function compiles and returns a Keras model.
    Should be passed to KerasClassifier in the Keras scikit-learn API.
    """

    model = Sequential()
    model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(28, 28,␣
↪1)))
    model.add(Conv2D(64, (3, 3), activation='relu'))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.5))
```

(continues on next page)

(continued from previous page)

```
    model.add(Dense(10, activation='softmax'))

    model.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=[
↪'accuracy'])

    return model
```

```
/home/namazu/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: FutureWarning:
↪ Conversion of the second argument of issubdtype from `float` to `np.floating` is␣
↪deprecated. In future, it will be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

For our purposes, the `classifier` which we will initialize now acts just like any scikit-learn estimator.

[2]:
```
# create the classifier
classifier = KerasClassifier(create_keras_model)
```

23.2 Active learning with Keras

In this example, we are going to use the famous MNIST dataset, which is available as a built-in for Keras.

[3]:
```
import numpy as np
from keras.datasets import mnist

# read training data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = X_train.reshape(60000, 28, 28, 1).astype('float32') / 255
X_test = X_test.reshape(10000, 28, 28, 1).astype('float32') / 255
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

# assemble initial data
n_initial = 1000
initial_idx = np.random.choice(range(len(X_train)), size=n_initial, replace=False)
X_initial = X_train[initial_idx]
y_initial = y_train[initial_idx]

# generate the pool
# remove the initial data from the training dataset
X_pool = np.delete(X_train, initial_idx, axis=0)[:5000]
y_pool = np.delete(y_train, initial_idx, axis=0)[:5000]
```

Active learning with data and classifier ready is as easy as always. Because training is *very* expensive in large neural networks, this time we are going to query the best 200 instances each time we measure the uncertainty of the pool.

[4]:
```
from modAL.models import ActiveLearner

# initialize ActiveLearner
learner = ActiveLearner(
    estimator=classifier,
    X_training=X_initial, y_training=y_initial,
    verbose=1
)
```

```
Epoch 1/1
1000/1000 [==============================] - 4s 4ms/step - loss: 1.5794 - acc: 0.4790
```

To make sure that you train only on newly queried labels, pass `only_new=True` to the `.teach()` method of the learner.

[5]:
```
# the active learning loop
n_queries = 10
for idx in range(n_queries):
    print('Query no. %d' % (idx + 1))
    query_idx, query_instance = learner.query(X_pool, n_instances=100, verbose=0)
    learner.teach(
        X=X_pool[query_idx], y=y_pool[query_idx], only_new=True,
        verbose=1
    )
    # remove queried instance from pool
    X_pool = np.delete(X_pool, query_idx, axis=0)
    y_pool = np.delete(y_pool, query_idx, axis=0)
```

```
Query no. 1
Epoch 1/1
100/100 [==============================] - 1s 10ms/step - loss: 2.0987 - acc: 0.3300
Query no. 2
Epoch 1/1
100/100 [==============================] - 1s 7ms/step - loss: 2.1222 - acc: 0.3300
Query no. 3
Epoch 1/1
100/100 [==============================] - 1s 8ms/step - loss: 2.0558 - acc: 0.4900
Query no. 4
Epoch 1/1
100/100 [==============================] - 1s 9ms/step - loss: 1.6943 - acc: 0.4700
Query no. 5
Epoch 1/1
100/100 [==============================] - 1s 12ms/step - loss: 1.5865 - acc: 0.6200
Query no. 6
Epoch 1/1
100/100 [==============================] - 1s 14ms/step - loss: 1.8714 - acc: 0.3500
Query no. 7
Epoch 1/1
100/100 [==============================] - 1s 14ms/step - loss: 1.3940 - acc: 0.6700
Query no. 8
Epoch 1/1
100/100 [==============================] - 1s 14ms/step - loss: 2.1033 - acc: 0.3200
Query no. 9
Epoch 1/1
100/100 [==============================] - 1s 11ms/step - loss: 1.5666 - acc: 0.6700
Query no. 10
Epoch 1/1
100/100 [==============================] - 1s 12ms/step - loss: 2.0238 - acc: 0.2700
```

CHAPTER 24

Pytorch models in modAL workflows

Thanks to Skorch API, you can seamlessly integrate Pytorch models into your modAL workflow. In this tutorial, we shall quickly introduce how to use Skorch API of Keras and we are going to see how to do active learning with it. More details on the Keras scikit-learn API can be found here.

The executable script for this example can be found here!

24.1 Skorch API

By default, a Pytorch model's interface differs from what is used for scikit-learn estimators. However, with the use of Skorch wrapper, it is possible to adapt your model.

[1]:
```
import torch
from torch import nn
from skorch import NeuralNetClassifier

# build class for the skorch API
class Torch_Model(nn.Module):
    def __init__(self,):
        super(Torch_Model, self).__init__()
        self.convs = nn.Sequential(
                                nn.Conv2d(1,32,3),
                                nn.ReLU(),
                                nn.Conv2d(32,64,3),
                                nn.ReLU(),
                                nn.MaxPool2d(2),
                                nn.Dropout(0.25)
        )
        self.fcs = nn.Sequential(
                                nn.Linear(12*12*64,128),
                                nn.ReLU(),
                                nn.Dropout(0.5),
                                nn.Linear(128,10),
```

(continues on next page)

(continued from previous page)

```
        )

    def forward(self, x):
        out = x
        out = self.convs(out)
        out = out.view(-1,12*12*64)
        out = self.fcs(out)
        return out
```

For our purposes, the `classifier` which we will initialize now acts just like any scikit-learn estimator.

[2]:
```
# create the classifier
device = "cuda" if torch.cuda.is_available() else "cpu"
classifier = NeuralNetClassifier(Torch_Model,
                                 criterion=nn.CrossEntropyLoss,
                                 optimizer=torch.optim.Adam,
                                 train_split=None,
                                 verbose=1,
                                 device=device)
```

24.2 Active learning with Pytorch

In this example, we are going to use the famous MNIST dataset, which is available as a built-in for PyTorch.

[ ]:
```
import numpy as np
from torch.utils.data import DataLoader
from torchvision.transforms import ToTensor
from torchvision.datasets import MNIST


mnist_data = MNIST('.', download=True, transform=ToTensor())
dataloader = DataLoader(mnist_data, shuffle=True, batch_size=60000)
X, y = next(iter(dataloader))

# read training data
X_train, X_test, y_train, y_test = X[:50000], X[50000:], y[:50000], y[50000:]
X_train = X_train.reshape(50000, 1, 28, 28)
X_test = X_test.reshape(10000, 1, 28, 28)

# assemble initial data
n_initial = 1000
initial_idx = np.random.choice(range(len(X_train)), size=n_initial, replace=False)
X_initial = X_train[initial_idx]
y_initial = y_train[initial_idx]

# generate the pool
# remove the initial data from the training dataset
X_pool = np.delete(X_train, initial_idx, axis=0)[:5000]
y_pool = np.delete(y_train, initial_idx, axis=0)[:5000]
```

```
0it [00:00, ?it/s]
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./MNIST/
↪raw/train-images-idx3-ubyte.gz
```

```
 97%|| 9584640/9912422 [00:15<00:00, 1777143.52it/s]
```

```
Extracting ./MNIST/raw/train-images-idx3-ubyte.gz
```

```

0it [00:00, ?it/s]
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./MNIST/
↪raw/train-labels-idx1-ubyte.gz
```

```

  0%|          | 0/28881 [00:00<?, ?it/s]
 57%|    | 16384/28881 [00:00<00:00, 62622.03it/s]
32768it [00:00, 41627.01it/s]
0it [00:00, ?it/s]
```

```
Extracting ./MNIST/raw/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./MNIST/raw/
↪t10k-images-idx3-ubyte.gz
```

```

  0%|          | 0/1648877 [00:00<?, ?it/s]
```

Active learning with data and classifier ready is as easy as always. Because training is *very* expensive in large neural networks, this time we are going to query the best 200 instances each time we measure the uncertainty of the pool.

[ ]:
```
from modAL.models import ActiveLearner

# initialize ActiveLearner
learner = ActiveLearner(
    estimator=classifier,
    X_training=X_initial, y_training=y_initial,
)
```

To make sure that you train only on newly queried labels, pass `only_new=True` to the `.teach()` method of the learner.

[ ]:
```
# the active learning loop
n_queries = 10
for idx in range(n_queries):
    print('Query no. %d' % (idx + 1))
    query_idx, query_instance = learner.query(X_pool, n_instances=100)
    learner.teach(
        X=X_pool[query_idx], y=y_pool[query_idx], only_new=True,
    )
    # remove queried instance from pool
    X_pool = np.delete(X_pool, query_idx, axis=0)
    y_pool = np.delete(y_pool, query_idx, axis=0)
```

[ ]:
```

```

CHAPTER 25

modAL.models

class modAL.models.**ActiveLearner**(*estimator: sklearn.base.BaseEstimator*, *query_strategy: Callable = <function uncertainty_sampling>*, *X_training: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix, None] = None*, *y_training: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix, None] = None*, *bootstrap_init: bool = False*, ***fit_kwargs*)

This class is an abstract model of a general active learning algorithm.

Parameters

- **estimator** – The estimator to be used in the active learning loop.
- **query_strategy** – Function providing the query strategy for the active learning loop, for instance, modAL.uncertainty.uncertainty_sampling.
- **X_training** – Initial training samples, if available.
- **y_training** – Initial training labels corresponding to initial training samples.
- **bootstrap_init** – If initial training data is available, bootstrapping can be done during the first training. Useful when building Committee models with bagging.
- ****fit_kwargs** – keyword arguments.

estimator
The estimator to be used in the active learning loop.

query_strategy
Function providing the query strategy for the active learning loop.

X_training
If the model hasn't been fitted yet it is None, otherwise it contains the samples which the model has been trained on. If provided, the method fit() of estimator is called during __init__()

y_training
The labels corresponding to X_training.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.ensemble import RandomForestClassifier
>>> from modAL.models import ActiveLearner
>>> iris = load_iris()
>>> # give initial training examples
>>> X_training = iris['data'][[0, 50, 100]]
>>> y_training = iris['target'][[0, 50, 100]]
>>>
>>> # initialize active learner
>>> learner = ActiveLearner(
...     estimator=RandomForestClassifier(),
...     X_training=X_training, y_training=y_training
... )
>>>
>>> # querying for labels
>>> query_idx, query_sample = learner.query(iris['data'])
>>>
>>> # ...obtaining new labels from the Oracle...
>>>
>>> # teaching newly labelled examples
>>> learner.teach(
...     X=iris['data'][query_idx].reshape(1, -1),
...     y=iris['target'][query_idx].reshape(1, )
... )
```

fit (*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], bootstrap: bool = False, **fit_kwargs*) → modAL.models.base.BaseLearner

Interface for the fit method of the predictor. Fits the predictor to the supplied data, then stores it internally for the active learning loop.

Parameters

- **X** – The samples to be fitted.
- **y** – The corresponding labels.
- **bootstrap** – If true, trains the estimator on a set bootstrapped from X. Useful for building Committee models with bagging.
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

Note: When using scikit-learn estimators, calling this method will make the ActiveLearner forget all training data it has seen!

Returns self

predict (*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_kwargs*) → Any

Estimator predictions for X. Interface with the predict method of the estimator.

Parameters

- **X** – The samples to be predicted.
- ****predict_kwargs** – Keyword arguments to be passed to the predict method of the estimator.

Returns Estimator predictions for X.

predict_proba(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → Any

Class probabilities if the predictor is a classifier. Interface with the predict_proba method of the classifier.

Parameters

- **X** – The samples for which the class probabilities are to be predicted.
- ****predict_proba_kwargs** – Keyword arguments to be passed to the predict_proba method of the classifier.

Returns Class probabilities for X.

query()

Finds the n_instances most informative point in the data provided by calling the query_strategy function.

Parameters

- ***query_args** – The arguments for the query strategy. For instance, in the case of *uncertainty_sampling()*, it is the pool of samples from which the query strategy should choose instances to request labels.
- ****query_kwargs** – Keyword arguments for the query strategy function.

Returns Value of the query_strategy function. Should be the indices of the instances from the pool chosen to be labelled and the instances themselves. Can be different in other cases, for instance only the instance to be labelled upon query synthesis.

score(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **score_kwargs*) → Any

Interface for the score method of the predictor.

Parameters

- **X** – The samples for which prediction accuracy is to be calculated.
- **y** – Ground truth labels for X.
- ****score_kwargs** – Keyword arguments to be passed to the .score() method of the predictor.

Returns The score of the predictor.

teach(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], bootstrap: bool = False, only_new: bool = False, **fit_kwargs*) → None

Adds X and y to the known training data and retrains the predictor with the augmented dataset.

Parameters

- **X** – The new samples for which the labels are supplied by the expert.
- **y** – Labels corresponding to the new instances in X.
- **bootstrap** – If True, training is done on a bootstrapped dataset. Useful for building Committee models with bagging.
- **only_new** – If True, the model is retrained using only X and y, ignoring the previously provided examples. Useful when working with models where the .fit() method doesn't retrain the model from scratch (e. g. in tensorflow or keras).
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

class modAL.models.**BayesianOptimizer**(*estimator: sklearn.base.BaseEstimator*, *query_strategy: Callable = <function max_EI>*, *X_training: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix, None] = None*, *y_training: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix, None] = None*, *bootstrap_init: bool = False*, ***fit_kwargs*)

This class is an abstract model of a Bayesian optimizer algorithm.

Parameters

- **estimator** – The estimator to be used in the Bayesian optimization. (For instance, a GaussianProcessRegressor.)
- **query_strategy** – Function providing the query strategy for Bayesian optimization, for instance, modAL.acquisitions.max_EI.
- **X_training** – Initial training samples, if available.
- **y_training** – Initial training labels corresponding to initial training samples.
- **bootstrap_init** – If initial training data is available, bootstrapping can be done during the first training. Useful when building Committee models with bagging.
- ****fit_kwargs** – keyword arguments.

estimator
The estimator to be used in the Bayesian optimization.

query_strategy
Function providing the query strategy for Bayesian optimization.

X_training
If the model hasn't been fitted yet it is None, otherwise it contains the samples which the model has been trained on.

y_training
The labels corresponding to X_training.

X_max
argmax of the function so far.

y_max
Max of the function so far.

Examples

```
>>> import numpy as np
>>> from functools import partial
>>> from sklearn.gaussian_process import GaussianProcessRegressor
>>> from sklearn.gaussian_process.kernels import Matern
>>> from modAL.models import BayesianOptimizer
>>> from modAL.acquisition import optimizer_PI, optimizer_EI, optimizer_UCB, max_
↪PI, max_EI, max_UCB
>>>
>>> # generating the data
>>> X = np.linspace(0, 20, 1000).reshape(-1, 1)
>>> y = np.sin(X)/2 - ((10 - X)**2)/50 + 2
>>>
>>> # assembling initial training set
>>> X_initial, y_initial = X[150].reshape(1, -1), y[150].reshape(1, -1)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # defining the kernel for the Gaussian process
>>> kernel = Matern(length_scale=1.0)
>>>
>>> tr = 0.1
>>> PI_tr = partial(optimizer_PI, tradeoff=tr)
>>> PI_tr.__name__ = 'PI, tradeoff = %1.1f' % tr
>>> max_PI_tr = partial(max_PI, tradeoff=tr)
>>>
>>> acquisitions = zip(
...     [PI_tr, optimizer_EI, optimizer_UCB],
...     [max_PI_tr, max_EI, max_UCB],
... )
>>>
>>> for acquisition, query_strategy in acquisitions:
...     # initializing the optimizer
...     optimizer = BayesianOptimizer(
...         estimator=GaussianProcessRegressor(kernel=kernel),
...         X_training=X_initial, y_training=y_initial,
...         query_strategy=query_strategy
...     )
...
...     for n_query in range(5):
...         # query
...         query_idx, query_inst = optimizer.query(X)
...         optimizer.teach(X[query_idx].reshape(1, -1), y[query_idx].reshape(1, -
↪1))
```

fit (*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], bootstrap: bool = False, **fit_kwargs*) → modAL.models.base.BaseLearner

Interface for the fit method of the predictor. Fits the predictor to the supplied data, then stores it internally for the active learning loop.

Parameters

- **X** – The samples to be fitted.
- **y** – The corresponding labels.
- **bootstrap** – If true, trains the estimator on a set bootstrapped from X. Useful for building Committee models with bagging.
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

Note: When using scikit-learn estimators, calling this method will make the ActiveLearner forget all training data it has seen!

Returns self

predict (*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_kwargs*) → Any

Estimator predictions for X. Interface with the predict method of the estimator.

Parameters

- **X** – The samples to be predicted.

- ****predict_kwargs** – Keyword arguments to be passed to the predict method of the estimator.

Returns Estimator predictions for X.

predict_proba(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → Any

Class probabilities if the predictor is a classifier. Interface with the predict_proba method of the classifier.

Parameters

- **X** – The samples for which the class probabilities are to be predicted.
- ****predict_proba_kwargs** – Keyword arguments to be passed to the predict_proba method of the classifier.

Returns Class probabilities for X.

query()

Finds the n_instances most informative point in the data provided by calling the query_strategy function.

Parameters

- ***query_args** – The arguments for the query strategy. For instance, in the case of *uncertainty_sampling()*, it is the pool of samples from which the query strategy should choose instances to request labels.
- ****query_kwargs** – Keyword arguments for the query strategy function.

Returns Value of the query_strategy function. Should be the indices of the instances from the pool chosen to be labelled and the instances themselves. Can be different in other cases, for instance only the instance to be labelled upon query synthesis.

score(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **score_kwargs*) → Any

Interface for the score method of the predictor.

Parameters

- **X** – The samples for which prediction accuracy is to be calculated.
- **y** – Ground truth labels for X.
- ****score_kwargs** – Keyword arguments to be passed to the .score() method of the predictor.

Returns The score of the predictor.

teach(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], bootstrap: bool = False, only_new: bool = False, **fit_kwargs*) → None

Adds X and y to the known training data and retrains the predictor with the augmented dataset. This method also keeps track of the maximum value encountered in the training data.

Parameters

- **X** – The new samples for which the values are supplied.
- **y** – Values corresponding to the new instances in X.
- **bootstrap** – If True, training is done on a bootstrapped dataset. Useful for building Committee models with bagging. (Default value = False)
- **only_new** – If True, the model is retrained using only X and y, ignoring the previously provided examples. Useful when working with models where the .fit() method doesn't retrain the model from scratch (for example, in tensorflow or keras).

- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

class modAL.models.**Committee**(*learner_list: List[modAL.models.learners.ActiveLearner], query_strategy: Callable = <function vote_entropy_sampling>*)

This class is an abstract model of a committee-based active learning algorithm.

Parameters

- **learner_list** – A list of ActiveLearners forming the Committee.
- **query_strategy** – Query strategy function. Committee supports disagreement-based query strategies from *modAL.disagreement*, but uncertainty-based ones from *modAL.uncertainty* are also supported.

classes_
Class labels known by the Committee.

n_classes_
Number of classes known by the Committee.

Examples

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.neighbors import KNeighborsClassifier
>>> from sklearn.ensemble import RandomForestClassifier
>>> from modAL.models import ActiveLearner, Committee
>>>
>>> iris = load_iris()
>>>
>>> # initialize ActiveLearners
>>> learner_1 = ActiveLearner(
...     estimator=RandomForestClassifier(),
...     X_training=iris['data'][[0, 50, 100]], y_training=iris['target'][[0, 50,␣
↪100]]
... )
>>> learner_2 = ActiveLearner(
...     estimator=KNeighborsClassifier(n_neighbors=3),
...     X_training=iris['data'][[1, 51, 101]], y_training=iris['target'][[1, 51,␣
↪101]]
... )
>>>
>>> # initialize the Committee
>>> committee = Committee(
...     learner_list=[learner_1, learner_2]
... )
>>>
>>> # querying for labels
>>> query_idx, query_sample = committee.query(iris['data'])
>>>
>>> # ...obtaining new labels from the Oracle...
>>>
>>> # teaching newly labelled examples
>>> committee.teach(
...     X=iris['data'][query_idx].reshape(1, -1),
...     y=iris['target'][query_idx].reshape(1, )
... )
```

fit(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **fit_kwargs*) → modAL.models.base.BaseCommittee

Fits every learner to a subset sampled with replacement from X. Calling this method makes the learner forget the data it has seen up until this point and replaces it with X! If you would like to perform bootstrapping on each learner using the data it has seen, use the method .rebag()!

Calling this method makes the learner forget the data it has seen up until this point and replaces it with X!

Parameters

- **X** – The samples to be fitted on.
- **y** – The corresponding labels.
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

predict(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → Any

Predicts the class of the samples by picking the consensus prediction.

Parameters

- **X** – The samples to be predicted.
- ****predict_proba_kwargs** – Keyword arguments to be passed to the *predict_proba()* of the Committee.

Returns The predicted class labels for X.

predict_proba(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → Any

Consensus probabilities of the Committee.

Parameters

- **X** – The samples for which the class probabilities are to be predicted.
- ****predict_proba_kwargs** – Keyword arguments to be passed to the *predict_proba()* of the Committee.

Returns Class probabilities for X.

query()

Finds the n_instances most informative point in the data provided by calling the query_strategy function.

Parameters

- ***query_args** – The arguments for the query strategy. For instance, in the case of *max_disagreement_sampling()*, it is the pool of samples from which the query. strategy should choose instances to request labels.
- ****query_kwargs** – Keyword arguments for the query strategy function.

Returns Return value of the query_strategy function. Should be the indices of the instances from the pool chosen to be labelled and the instances themselves. Can be different in other cases, for instance only the instance to be labelled upon query synthesis.

rebag(***fit_kwargs*) → None

Refits every learner with a dataset bootstrapped from its training instances. Contrary to .bag(), it bootstraps the training data for each learner based on its own examples.

Parameters ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

score(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], sample_weight: List[float] = None*) → Any

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

Parameters

- **X** – The samples to score.
- **y** – Ground truth labels corresponding to X.
- **sample_weight** – Sample weights.

Returns Mean accuracy of the classifiers.

teach(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], bootstrap: bool = False, only_new: bool = False, **fit_kwargs*) → None

Adds X and y to the known training data for each learner and retrains learners with the augmented dataset.

Parameters

- **X** – The new samples for which the labels are supplied by the expert.
- **y** – Labels corresponding to the new instances in X.
- **bootstrap** – If True, trains each learner on a bootstrapped set. Useful when building the ensemble by bagging.
- **only_new** – If True, the model is retrained using only X and y, ignoring the previously provided examples.
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

vote(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_kwargs*) → Any

Predicts the labels for the supplied data for each learner in the Committee.

Parameters

- **X** – The samples to cast votes.
- ****predict_kwargs** – Keyword arguments to be passed to the *predict()* of the learners.

Returns The predicted class for each learner in the Committee and each sample in X.

vote_proba(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → Any

Predicts the probabilities of the classes for each sample and each learner.

Parameters

- **X** – The samples for which class probabilities are to be calculated.
- ****predict_proba_kwargs** – Keyword arguments for the *predict_proba()* of the learners.

Returns Probabilities of each class for each learner and each instance.

class modAL.models.**CommitteeRegressor**(*learner_list: List[modAL.models.learners.ActiveLearner], query_strategy: Callable = <function max_std_sampling>*)

This class is an abstract model of a committee-based active learning regression.

Parameters

- **learner_list** – A list of ActiveLearners forming the CommitteeRegressor.
- **query_strategy** – Query strategy function.

Examples

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.gaussian_process import GaussianProcessRegressor
>>> from sklearn.gaussian_process.kernels import WhiteKernel, RBF
>>> from modAL.models import ActiveLearner, CommitteeRegressor
>>>
>>> # generating the data
>>> X = np.concatenate((np.random.rand(100)-1, np.random.rand(100)))
>>> y = np.abs(X) + np.random.normal(scale=0.2, size=X.shape)
>>>
>>> # initializing the regressors
>>> n_initial = 10
>>> kernel = RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e3)) +␣
↪WhiteKernel(noise_level=1, noise_level_bounds=(1e-10, 1e+1))
>>>
>>> initial_idx = list()
>>> initial_idx.append(np.random.choice(range(100), size=n_initial,␣
↪replace=False))
>>> initial_idx.append(np.random.choice(range(100, 200), size=n_initial,␣
↪replace=False))
>>> learner_list = [ActiveLearner(
...                         estimator=GaussianProcessRegressor(kernel),
...                         X_training=X[idx].reshape(-1, 1), y_training=y[idx].
↪reshape(-1, 1)
...                 )
...                 for idx in initial_idx]
>>>
>>> # query strategy for regression
>>> def ensemble_regression_std(regressor, X):
...     _, std = regressor.predict(X, return_std=True)
...     query_idx = np.argmax(std)
...     return query_idx, X[query_idx]
>>>
>>> # initializing the CommitteeRegressor
>>> committee = CommitteeRegressor(
...     learner_list=learner_list,
...     query_strategy=ensemble_regression_std
... )
>>>
>>> # active regression
>>> n_queries = 10
>>> for idx in range(n_queries):
...     query_idx, query_instance = committee.query(X.reshape(-1, 1))
...     committee.teach(X[query_idx].reshape(-1, 1), y[query_idx].reshape(-1, 1))
```

fit(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]*, *y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]*, ***fit_kwargs*) → modAL.models.base.BaseCommittee

Fits every learner to a subset sampled with replacement from X. Calling this method makes the learner forget the data it has seen up until this point and replaces it with X! If you would like to perform bootstrapping on each learner using the data it has seen, use the method .rebag()!

Calling this method makes the learner forget the data it has seen up until this point and replaces it with X!

Parameters

- **X** – The samples to be fitted on.

- **y** – The corresponding labels.
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

predict(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], return_std: bool = False, **predict_kwargs*) → Any

Predicts the values of the samples by averaging the prediction of each regressor.

Parameters

- **X** – The samples to be predicted.
- ****predict_kwargs** – Keyword arguments to be passed to the *vote()* method of the CommitteeRegressor.

Returns The predicted class labels for X.

query()

Finds the n_instances most informative point in the data provided by calling the query_strategy function.

Parameters

- ***query_args** – The arguments for the query strategy. For instance, in the case of *max_disagreement_sampling()*, it is the pool of samples from which the query. strategy should choose instances to request labels.
- ****query_kwargs** – Keyword arguments for the query strategy function.

Returns Return value of the query_strategy function. Should be the indices of the instances from the pool chosen to be labelled and the instances themselves. Can be different in other cases, for instance only the instance to be labelled upon query synthesis.

rebag(***fit_kwargs*) → None

Refits every learner with a dataset bootstrapped from its training instances. Contrary to .bag(), it bootstraps the training data for each learner based on its own examples.

Parameters ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

teach(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], y: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], bootstrap: bool = False, only_new: bool = False, **fit_kwargs*) → None

Adds X and y to the known training data for each learner and retrains learners with the augmented dataset.

Parameters

- **X** – The new samples for which the labels are supplied by the expert.
- **y** – Labels corresponding to the new instances in X.
- **bootstrap** – If True, trains each learner on a bootstrapped set. Useful when building the ensemble by bagging.
- **only_new** – If True, the model is retrained using only X and y, ignoring the previously provided examples.
- ****fit_kwargs** – Keyword arguments to be passed to the fit method of the predictor.

vote(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_kwargs*)

Predicts the values for the supplied data for each regressor in the CommitteeRegressor.

Parameters

- **X** – The samples to cast votes.

- ****predict_kwargs** – Keyword arguments to be passed to *predict()* of the learners.

Returns The predicted value for each regressor in the CommitteeRegressor and each sample in X.

CHAPTER 26

modAL.uncertainty

Uncertainty measures and uncertainty based sampling strategies for the active learning models.

modAL.uncertainty.**classifier_entropy**(*classifier: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → numpy.ndarray

Entropy of predictions of the for the provided samples.

Parameters

- **classifier** – The classifier for which the prediction entropy is to be measured.
- **X** – The samples for which the prediction entropy is to be measured.
- ****predict_proba_kwargs** – Keyword arguments to be passed for the `predict_proba()` of the classifier.

Returns Entropy of the class probabilities.

modAL.uncertainty.**classifier_margin**(*classifier: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → numpy.ndarray

Classification margin uncertainty of the classifier for the provided samples. This uncertainty measure takes the first and second most likely predictions and takes the difference of their probabilities, which is the margin.

Parameters

- **classifier** – The classifier for which the prediction margin is to be measured.
- **X** – The samples for which the prediction margin of classification is to be measured.
- ****predict_proba_kwargs** – Keyword arguments to be passed for the `predict_proba()` of the classifier.

Returns Margin uncertainty, which is the difference of the probabilities of first and second most likely predictions.

modAL.uncertainty.**classifier_uncertainty**(*classifier: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → numpy.ndarray

Classification uncertainty of the classifier for the provided samples.

Parameters

- **classifier** – The classifier for which the uncertainty is to be measured.
- **X** – The samples for which the uncertainty of classification is to be measured.
- ****predict_proba_kwargs** – Keyword arguments to be passed for the `predict_proba()` of the classifier.

Returns Classifier uncertainty, which is 1 - P(prediction is correct).

modAL.uncertainty.**entropy_sampling**(*classifier: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False, **uncertainty_measure_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Entropy sampling query strategy. Selects the instances where the class probabilities have the largest entropy.

Parameters

- **classifier** – The classifier for which the labels are to be queried.
- **X** – The pool of samples to query from.
- **n_instances** – Number of samples to be queried.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.
- ****uncertainty_measure_kwargs** – Keyword arguments to be passed for the uncertainty measure function.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.uncertainty.**margin_sampling**(*classifier: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False, **uncertainty_measure_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Margin sampling query strategy. Selects the instances where the difference between the first most likely and second most likely classes are the smallest. :param classifier: The classifier for which the labels are to be queried. :param X: The pool of samples to query from. :param n_instances: Number of samples to be queried. :param random_tie_break: If True, shuffles utility scores to randomize the order. This

can be used to break the tie when the highest utility score is not unique.

Parameters ****uncertainty_measure_kwargs** – Keyword arguments to be passed for the uncertainty measure function.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.uncertainty.**uncertainty_sampling**(*classifier: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False, **uncertainty_measure_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Uncertainty sampling query strategy. Selects the least sure instances for labelling.

Parameters

- **classifier** – The classifier for which the labels are to be queried.
- **X** – The pool of samples to query from.
- **n_instances** – Number of samples to be queried.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.
- ****uncertainty_measure_kwargs** – Keyword arguments to be passed for the uncertainty measure function.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

CHAPTER 27

modAL.disagreement

Disagreement measures and disagreement based query strategies for the Committee model.

modAL.disagreement.**KL_max_disagreement**(*committee: modAL.models.base.BaseCommittee, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → numpy.ndarray

Calculates the max disagreement for the Committee. First it computes the class probabilties of X for each learner in the Committee, then calculates the consensus probability distribution by averaging the individual class probabilities for each learner. Then each learner's class probabilities are compared to the consensus distribution in the sense of Kullback-Leibler divergence. The max disagreement for a given sample is the argmax of the KL divergences of the learners from the consensus probability.

Parameters

- **committee** – The modAL.models.BaseCommittee instance for which the max disagreement is to be calculated.
- **X** – The data for which the max disagreement is to be calculated.
- ****predict_proba_kwargs** – Keyword arguments for the predict_proba() of the Committee.

Returns Max disagreement of the Committee for the samples in X.

modAL.disagreement.**consensus_entropy**(*committee: modAL.models.base.BaseCommittee, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → numpy.ndarray

Calculates the consensus entropy for the Committee. First it computes the class probabilties of X for each learner in the Committee, then calculates the consensus probability distribution by averaging the individual class probabilities for each learner. The entropy of the consensus probability distribution is the vote entropy of the Committee, which is returned.

Parameters

- **committee** – The modAL.models.BaseCommittee instance for which the consensus entropy is to be calculated.

- **X** – The data for which the consensus entropy is to be calculated.
- ****predict_proba_kwargs** – Keyword arguments for the `predict_proba()` of the Committee.

Returns Consensus entropy of the Committee for the samples in X.

`modAL.disagreement.`**`consensus_entropy_sampling`**(*committee: modAL.models.base.BaseCommittee, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break=False, **disagreement_measure_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Consensus entropy sampling strategy.

Parameters

- **committee** – The committee for which the labels are to be queried.
- **X** – The pool of samples to query from.
- **n_instances** – Number of samples to be queried.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.
- ****disagreement_measure_kwargs** – Keyword arguments to be passed for the disagreement measure function.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

`modAL.disagreement.`**`max_disagreement_sampling`**(*committee: modAL.models.base.BaseCommittee, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break=False, **disagreement_measure_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Maximum disagreement sampling strategy.

Parameters

- **committee** – The committee for which the labels are to be queried.
- **X** – The pool of samples to query from.
- **n_instances** – Number of samples to be queried.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.
- ****disagreement_measure_kwargs** – Keyword arguments to be passed for the disagreement measure function.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.disagreement.**max_std_sampling**(*regressor: sklearn.base.BaseEstimator, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break=False, **predict_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Regressor standard deviation sampling strategy.

Parameters

- **regressor** – The regressor for which the labels are to be queried.
- **X** – The pool of samples to query from.
- **n_instances** – Number of samples to be queried.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.
- ****predict_kwargs** – Keyword arguments to be passed to `predict()` of the CommiteeRegressor.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.disagreement.**vote_entropy**(*committee: modAL.models.base.BaseCommittee, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], **predict_proba_kwargs*) → numpy.ndarray

Calculates the vote entropy for the Committee. First it computes the predictions of X for each learner in the Committee, then calculates the probability distribution of the votes. The entropy of this distribution is the vote entropy of the Committee, which is returned.

Parameters

- **committee** – The `modAL.models.BaseCommittee` instance for which the vote entropy is to be calculated.
- **X** – The data for which the vote entropy is to be calculated.
- ****predict_proba_kwargs** – Keyword arguments for the `predict_proba()` of the Committee.

Returns Vote entropy of the Committee for the samples in X.

modAL.disagreement.**vote_entropy_sampling**(*committee: modAL.models.base.BaseCommittee, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break=False, **disagreement_measure_kwargs*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Vote entropy sampling strategy.

Parameters

- **committee** – The committee for which the labels are to be queried.
- **X** – The pool of samples to query from.
- **n_instances** – Number of samples to be queried.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

- **`**disagreement_measure_kwargs`** – Keyword arguments to be passed for the disagreement measure function.

Returns

The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

CHAPTER 28

modAL.multilabel

modAL.multilabel.**SVM_binary_minimum**(*classifier: modAL.models.learners.ActiveLearner, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

SVM binary minimum multilabel active learning strategy. For details see the paper Klaus Brinker, On Active Learning in Multi-label Classification (https://link.springer.com/chapter/10.1007%2F3-540-31314-1_24)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried. Must be an SVM model such as the ones from sklearn.svm.
- **X_pool** – The pool of samples to query from.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

modAL.multilabel.**avg_confidence**(*classifier: sklearn.multiclass.OneVsRestClassifier, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

AvgConfidence query strategy for multilabel classification.

For more details on this query strategy, see Esuli and Sebastiani., Active Learning Strategies for Multi-Label Text Classification (http://dx.doi.org/10.1007/978-3-642-00958-7_12)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried.
- **X_pool** – The pool of samples to query from.

- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

modAL.multilabel.**avg_score**(*classifier: sklearn.multiclass.OneVsRestClassifier, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

AvgScore query strategy for multilabel classification.

For more details on this query strategy, see Esuli and Sebastiani., Active Learning Strategies for Multi-Label Text Classification (http://dx.doi.org/10.1007/978-3-642-00958-7_12)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried.
- **X_pool** – The pool of samples to query from.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

modAL.multilabel.**max_loss**(*classifier: sklearn.multiclass.OneVsRestClassifier, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Max Loss query strategy for SVM multilabel classification.

For more details on this query strategy, see Li et al., Multilabel SVM active learning for image classification (http://dx.doi.org/10.1109/ICIP.2004.1421535)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried. Should be an SVM model such as the ones from sklearn.svm. Although the function will execute for other models as well, the mathematical calculations in Li et al. work only for SVM-s.
- **X_pool** – The pool of samples to query from.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

modAL.multilabel.**max_score**(*classifier: sklearn.multiclass.OneVsRestClassifier, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = 1*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

MaxScore query strategy for multilabel classification.

For more details on this query strategy, see Esuli and Sebastiani., Active Learning Strategies for Multi-Label Text Classification (http://dx.doi.org/10.1007/978-3-642-00958-7_12)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried.
- **X_pool** – The pool of samples to query from.

- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

modAL.multilabel.**mean_max_loss**(*classifier: sklearn.multiclass.OneVsRestClassifier, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Mean Max Loss query strategy for SVM multilabel classification.

For more details on this query strategy, see Li et al., Multilabel SVM active learning for image classification (http://dx.doi.org/10.1109/ICIP.2004.1421535)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried. Should be an SVM model such as the ones from sklearn.svm. Although the function will execute for other models as well, the mathematical calculations in Li et al. work only for SVM-s.
- **X_pool** – The pool of samples to query from.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

modAL.multilabel.**min_confidence**(*classifier: sklearn.multiclass.OneVsRestClassifier, X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 1, random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

MinConfidence query strategy for multilabel classification.

For more details on this query strategy, see Esuli and Sebastiani., Active Learning Strategies for Multi-Label Text Classification (http://dx.doi.org/10.1007/978-3-642-00958-7_12)

Parameters

- **classifier** – The multilabel classifier for which the labels are to be queried.
- **X_pool** – The pool of samples to query from.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The index of the instance from X_pool chosen to be labelled; the instance from X_pool chosen to be labelled.

CHAPTER 29

modAL.expected_error

Expected error reduction framework for active learning.

modAL.expected_error.**expected_error_reduction**(*learner: modAL.models.learners.ActiveLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], loss: str = 'binary', p_subsample: float = 1.0, n_instances: int = 1, random_tie_break: bool = False*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Expected error reduction query strategy.

References

Roy and McCallum, 2001 (http://groups.csail.mit.edu/rrg/papers/icml01.pdf)

Parameters

- **learner** – The ActiveLearner object for which the expected error is to be estimated.
- **X** – The samples.
- **loss** – The loss function to be used. Can be 'binary' or 'log'.
- **p_subsample** – Probability of keeping a sample from the pool when calculating expected error. Significantly improves runtime for large sample pools.
- **n_instances** – The number of instances to be sampled.
- **random_tie_break** – If True, shuffles utility scores to randomize the order. This can be used to break the tie when the highest utility score is not unique.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

CHAPTER 30

modAL.acquisition

Acquisition functions for Bayesian optimization.

modAL.acquisition.**max_EI**(*optimizer: modAL.models.base.BaseLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], tradeoff: float = 0, n_instances: int = 1*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Maximum EI query strategy. Selects the instance with highest expected improvement.

Parameters

- **optimizer** – The *BayesianOptimizer* object for which the utility is to be calculated.
- **X** – The samples for which the expected improvement is to be calculated.
- **tradeoff** – Value controlling the tradeoff parameter.
- **n_instances** – Number of samples to be queried.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.acquisition.**max_PI**(*optimizer: modAL.models.base.BaseLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], tradeoff: float = 0, n_instances: int = 1*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Maximum PI query strategy. Selects the instance with highest probability of improvement.

Parameters

- **optimizer** – The *BayesianOptimizer* object for which the utility is to be calculated.
- **X** – The samples for which the probability of improvement is to be calculated.
- **tradeoff** – Value controlling the tradeoff parameter.
- **n_instances** – Number of samples to be queried.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.acquisition.**max_UCB**(*optimizer: modAL.models.base.BaseLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], beta: float = 1, n_instances: int = 1*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Maximum UCB query strategy. Selects the instance with highest upper confidence bound.

Parameters

- **optimizer** – The *BayesianOptimizer* object for which the utility is to be calculated.
- **X** – The samples for which the maximum upper confidence bound is to be calculated.
- **beta** – Value controlling the beta parameter.
- **n_instances** – Number of samples to be queried.

Returns The indices of the instances from X chosen to be labelled; the instances from X chosen to be labelled.

modAL.acquisition.**optimizer_EI**(*optimizer: modAL.models.base.BaseLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], tradeoff: float = 0*) → numpy.ndarray

Expected improvement acquisition function for Bayesian optimization.

Parameters

- **optimizer** – The *BayesianOptimizer* object for which the utility is to be calculated.
- **X** – The samples for which the expected improvement is to be calculated.
- **tradeoff** – Value controlling the tradeoff parameter.

Returns Expected improvement utility score.

modAL.acquisition.**optimizer_PI**(*optimizer: modAL.models.base.BaseLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], tradeoff: float = 0*) → numpy.ndarray

Probability of improvement acquisition function for Bayesian optimization.

Parameters

- **optimizer** – The *BayesianOptimizer* object for which the utility is to be calculated.
- **X** – The samples for which the probability of improvement is to be calculated.
- **tradeoff** – Value controlling the tradeoff parameter.

Returns Probability of improvement utility score.

modAL.acquisition.**optimizer_UCB**(*optimizer: modAL.models.base.BaseLearner, X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], beta: float = 1*) → numpy.ndarray

Upper confidence bound acquisition function for Bayesian optimization.

Parameters

- **optimizer** – The *BayesianOptimizer* object for which the utility is to be calculated.
- **X** – The samples for which the upper confidence bound is to be calculated.
- **beta** – Value controlling the beta parameter.

Returns Upper confidence bound utility score.

CHAPTER 31

modAL.batch

Uncertainty measures that explicitly support batch-mode sampling for active learning models.

modAL.batch.**ranked_batch**(*classifier: Union[modAL.models.base.BaseLearner, modAL.models.base.BaseCommittee], unlabeled: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], uncertainty_scores: numpy.ndarray, n_instances: int, metric: Union[str, Callable], n_jobs: Optional[int]*) → numpy.ndarray

Query our top :n_instances: to request for labeling.

Refer to Cardoso et al.'s "Ranked batch-mode active learning": https://www.sciencedirect.com/science/article/pii/S0020025516313949

Parameters

- **classifier** – One of modAL's supported active learning models.
- **unlabeled** – Set of records to be considered for our active learning model.
- **uncertainty_scores** – Our classifier's predictions over the response variable.
- **n_instances** – Limit on the number of records to query from our unlabeled set.
- **metric** – This parameter is passed to `pairwise_distances()`.
- **n_jobs** – This parameter is passed to `pairwise_distances()`.

Returns The indices of the top n_instances ranked unlabelled samples.

modAL.batch.**select_cold_start_instance**(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], metric: Union[str, Callable], n_jobs: Optional[int]*) → Tuple[int, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Define what to do if our batch-mode sampling doesn't have any labeled data – a cold start.

If our ranked batch sampling algorithm doesn't have any labeled data to determine similarity among the uncertainty set, this function finds the element with highest average similarity to cold-start the batch selection.

Refer to Cardoso et al.'s "Ranked batch-mode active learning": https://www.sciencedirect.com/science/article/pii/S0020025516313949

Parameters

- **X** – The set of unlabeled records.
- **metric** – This parameter is passed to `pairwise_distances()`.
- **n_jobs** – This parameter is passed to `pairwise_distances()`.

Returns Index of the best cold-start instance from *X* chosen to be labelled; record of the best cold-start instance from *X* chosen to be labelled.

modAL.batch.**select_instance**(*X_training: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], X_pool: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], X_uncertainty: numpy.ndarray, mask: numpy.ndarray, metric: Union[str, Callable], n_jobs: Optional[int]*) → Tuple[numpy.ndarray, Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], numpy.ndarray]

Core iteration strategy for selecting another record from our unlabeled records.

Given a set of labeled records (X_training) and unlabeled records (X_pool) with uncertainty scores (X_uncertainty), we'd like to identify the best instance in X_pool that best balances uncertainty and dissimilarity.

Refer to Cardoso et al.'s "Ranked batch-mode active learning": https://www.sciencedirect.com/science/article/pii/S0020025516313949

Parameters

- **X_training** – Mix of both labeled and unlabeled records.
- **X_pool** – Unlabeled records to be selected for labeling.
- **X_uncertainty** – Uncertainty scores for unlabeled records to be selected for labeling.
- **mask** – Mask to exclude previously selected instances from the pool.
- **metric** – This parameter is passed to `pairwise_distances()`.
- **n_jobs** – This parameter is passed to `pairwise_distances()`.

Returns Index of the best index from X chosen to be labelled; a single record from our unlabeled set that is considered the most optimal incremental record for including in our query set.

modAL.batch.**uncertainty_batch_sampling**(*classifier: Union[modAL.models.base.BaseLearner, modAL.models.base.BaseCommittee], X: Union[numpy.ndarray, scipy.sparse.csr.csr_matrix], n_instances: int = 20, metric: Union[str, Callable] = 'euclidean', n_jobs: Optional[int] = None, **uncertainty_measure_kwargs*) → Tuple[numpy.ndarray, Union[numpy.ndarray, scipy.sparse.csr.csr_matrix]]

Batch sampling query strategy. Selects the least sure instances for labelling.

This strategy differs from *uncertainty_sampling()* because, although it is supported, traditional active learning query strategies suffer from sub-optimal record selection when passing *n_instances* > 1. This sampling strategy extends the interactive uncertainty query sampling by allowing for batch-mode uncertainty query sampling. Furthermore, it also enforces a ranking – that is, which records among the batch are most important for labeling?

Refer to Cardoso et al.'s "Ranked batch-mode active learning": https://www.sciencedirect.com/science/article/pii/S0020025516313949

Parameters

- **`classifier`** – One of modAL's supported active learning models.
- **`X`** – Set of records to be considered for our active learning model.
- **`n_instances`** – Number of records to return for labeling from *X*.
- **`metric`** – This parameter is passed to `pairwise_distances()`
- **`n_jobs`** – If not set, `pairwise_distances_argmin_min()` is used for calculation of distances between samples. Otherwise it is passed to `pairwise_distances()`.
- **`**uncertainty_measure_kwargs`** – Keyword arguments to be passed for the `predict_proba()` of the classifier.

Returns Indices of the instances from *X* chosen to be labelled; records from *X* chosen to be labelled.

CHAPTER 32

modAL.density

Measures for estimating the information density of a given sample.

modAL.density.**information_density**(*X: Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix], metric: Union[str, Callable] = 'euclidean'*) → numpy.ndarray

Calculates the information density metric of the given data using the given metric.

Parameters

- **X** – The data for which the information density is to be calculated.
- **metric** – The metric to be used. Should take two 1d numpy.ndarrays for argument.

Returns The information density for each sample.

modAL.density.**similarize_distance**(*distance_measure: Callable*) → Callable

Takes a distance measure and converts it into a information_density measure.

Parameters **distance_measure** – The distance measure to be converted into information_density measure.

Returns The information_density measure obtained from the given distance measure.

CHAPTER 33

modAL.utils

modAL.utils.**make_linear_combination**(**functions*, *weights: Optional[Sequence] = None*) → Callable

Takes the given functions and makes a function which returns the linear combination of the output of original functions. It works well with functions returning numpy arrays of the same shape.

Parameters

- ***functions** – Base functions for the linear combination.The functions shall have the same argument and if they return numpy arrays, the returned arrays shall have the same shape.
- **weights** – Coefficients of the functions in the linear combination. The i-th given function will be multiplied with weights[i].

Returns A function which returns the linear combination of the given functions output.

modAL.utils.**make_product**(**functions*, *exponents: Optional[Sequence] = None*) → Callable

Takes the given functions and makes a function which returns the product of the output of original functions. It works well with functions returning numpy arrays of the same shape.

Parameters

- ***functions** – Base functions for the product. The functions shall have the same argument and if they return numpy arrays, the returned arrays shall have the same shape.
- **exponents** – Exponents of the functions in the product. The i-th given function in the product will be raised to the power of exponents[i].

Returns A function which returns the product function of the given functions output.

modAL.utils.**make_query_strategy**(*utility_measure: Callable*, *selector: Callable*) → Callable

Takes the given utility measure and selector functions and makes a query strategy by combining them.

Parameters

- **utility_measure** – Utility measure, for instance *vote_entropy()*, but it can be a custom function as well. Should take a classifier and the unlabelled data and should return an array containing the utility scores.
- **selector** – Function selecting instances for query. Should take an array of utility scores and should return an array containing the queried items.

Returns A function which returns queried instances given a classifier and an unlabelled pool.

modAL.utils.**data_vstack**(*blocks: Container*) → Union[list, numpy.ndarray, scipy.sparse.csr.csr_matrix]

Stack vertically both sparse and dense arrays.

Parameters **blocks** – Sequence of modALinput objects.

Returns New sequence of vertically stacked elements.

modAL.utils.**multi_argmax**(*values: numpy.ndarray*, *n_instances: int = 1*) → numpy.ndarray

Selects the indices of the n_instances highest values.

Parameters

- **values** – Contains the values to be selected from.
- **n_instances** – Specifies how many indices to return.

Returns The indices of the n_instances largest values.

modAL.utils.**weighted_random**(*weights: numpy.ndarray*, *n_instances: int = 1*) → numpy.ndarray

Returns n_instances indices based on the weights.

Parameters

- **weights** – Contains the weights of the sampling.
- **n_instances** – Specifies how many indices to return.

Returns n_instances random indices based on the weights.

modAL.utils.**check_class_labels**(**args*) → bool

Checks the known class labels for each classifier.

Parameters ***args** – Classifier objects to check the known class labels.

Returns True, if class labels match for all classifiers, False otherwise.

modAL.utils.**check_class_proba**(*proba: numpy.ndarray*, *known_labels: Sequence*, *all_labels: Sequence*) → numpy.ndarray

Checks the class probabilities and reshapes it if not all labels are present in the classifier.

Parameters

- **proba** – The class probabilities of a classifier.
- **known_labels** – The class labels known by the classifier.
- **all_labels** – All class labels.

Returns Class probabilities augmented such that the probability of all classes is present. If the classifier is unaware of a particular class, all probabilities are zero.

Python Module Index

m

Index

N

O

P

Q

R

S

T

U

V

W

X

Y