
ABLkit

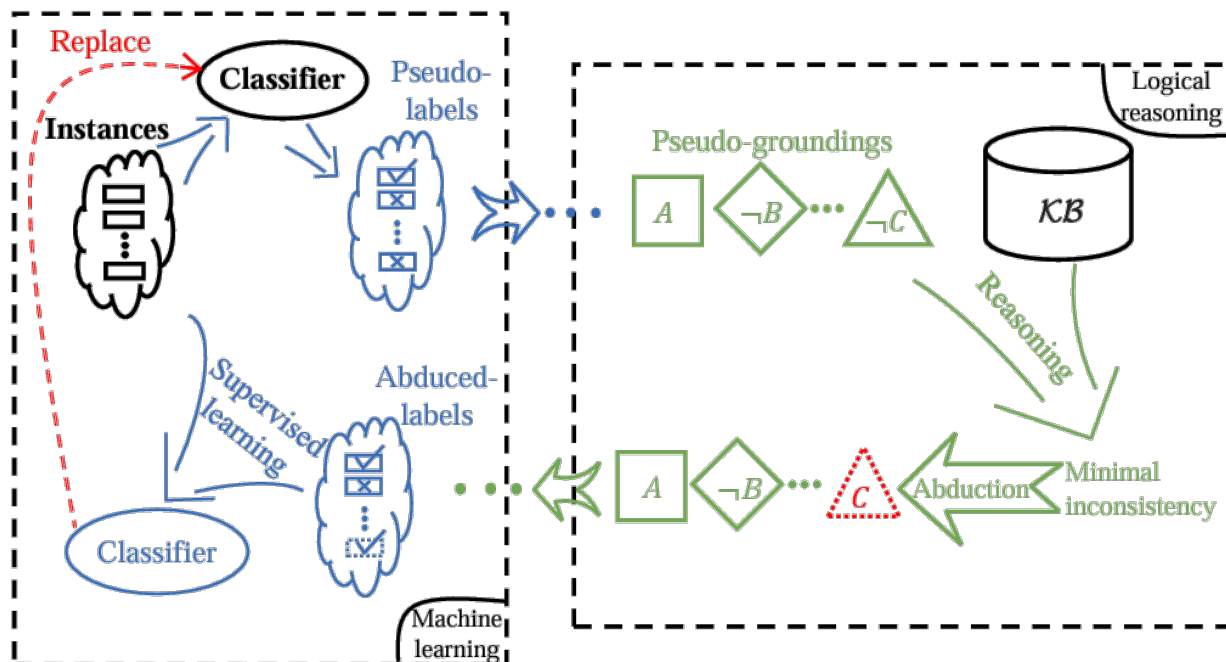
unknown

Mar 28, 2024

OVERVIEW

1	Installation	3
2	References	5
	Python Module Index	69
	Index	71

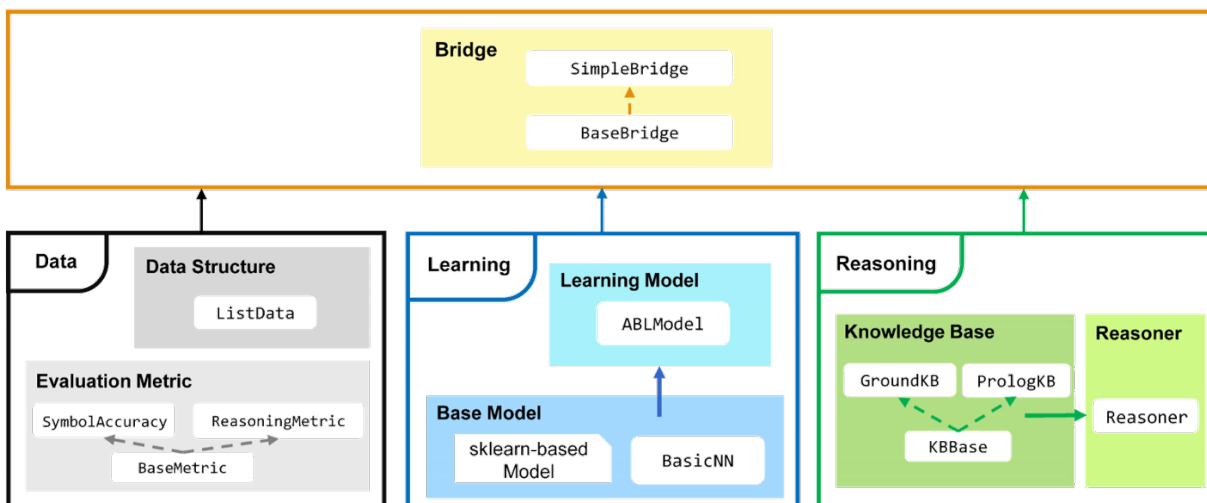
ABL is a novel paradigm that integrates machine learning and logical reasoning in a unified framework. It is suitable for tasks where both data and (logical) domain knowledge are available.



Key Features of ABLkit:

- **High Flexibility:** Compatible with various machine learning modules and logical reasoning components.
- **Easy-to-Use Interface:** Provide **data**, **model**, and **knowledge**, and get started with just a few lines of code.
- **Optimized Performance:** Optimization for high performance and accelerated training speed.

ABLkit encapsulates advanced ABL techniques, providing users with an efficient and convenient toolkit to develop dual-driven ABL systems, which leverage the power of both data and knowledge.



INSTALLATION

1.1 Install from PyPI

The easiest way to install ABLkit is using `pip`:

```
pip install ablkit
```

1.2 Install from Source

Alternatively, to install from source code, sequentially run following commands in your terminal/command line.

```
git clone https://github.com/AbductiveLearning/ABLkit.git
cd ABLkit
pip install -v -e .
```

1.3 (Optional) Install SWI-Prolog

If the use of a *Prolog-based knowledge base* is necessary, the installation of *SWI-Prolog* is also required:

For Linux users:

```
sudo apt-get install swi-prolog
```

For Windows and Mac users, please refer to the *SWI-Prolog Install Guide*.

REFERENCES

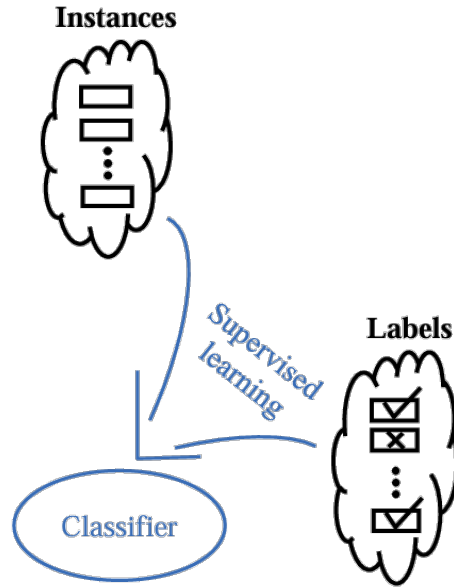
For more information about ABL, please refer to: [Zhou, 2019](#) and [Zhou and Huang, 2022](#).

```
@article{zhou2019abductive,
  title   = {Abductive learning: towards bridging machine learning and logical
↪reasoning},
  author  = {Zhou, Zhi-Hua},
  journal = {Science China Information Sciences},
  volume  = {62},
  number  = {7},
  pages   = {76101},
  year    = {2019}
}

@incollection{zhou2022abductive,
  title   = {Abductive Learning},
  author  = {Zhou, Zhi-Hua and Huang, Yu-Xuan},
  booktitle = {Neuro-Symbolic Artificial Intelligence: The State of the Art},
  editor   = {Pascal Hitzler and Md. Kamruzzaman Sarker},
  publisher = {{IOS} Press},
  pages    = {353--369},
  address  = {Amsterdam},
  year     = {2022}
}
```

2.1 Abductive Learning

Traditional supervised machine learning, e.g. classification, is predominantly data-driven, as shown in the below figure. Here, a set of data examples is given, including training instances $\{x_1, \dots, x_m\}$ and corresponding ground-truth labels $\{\text{label}(x_1), \dots, \text{label}(x_m)\}$. These data are then used to train a classifier model f , aiming to accurately predict the unseen data instances.



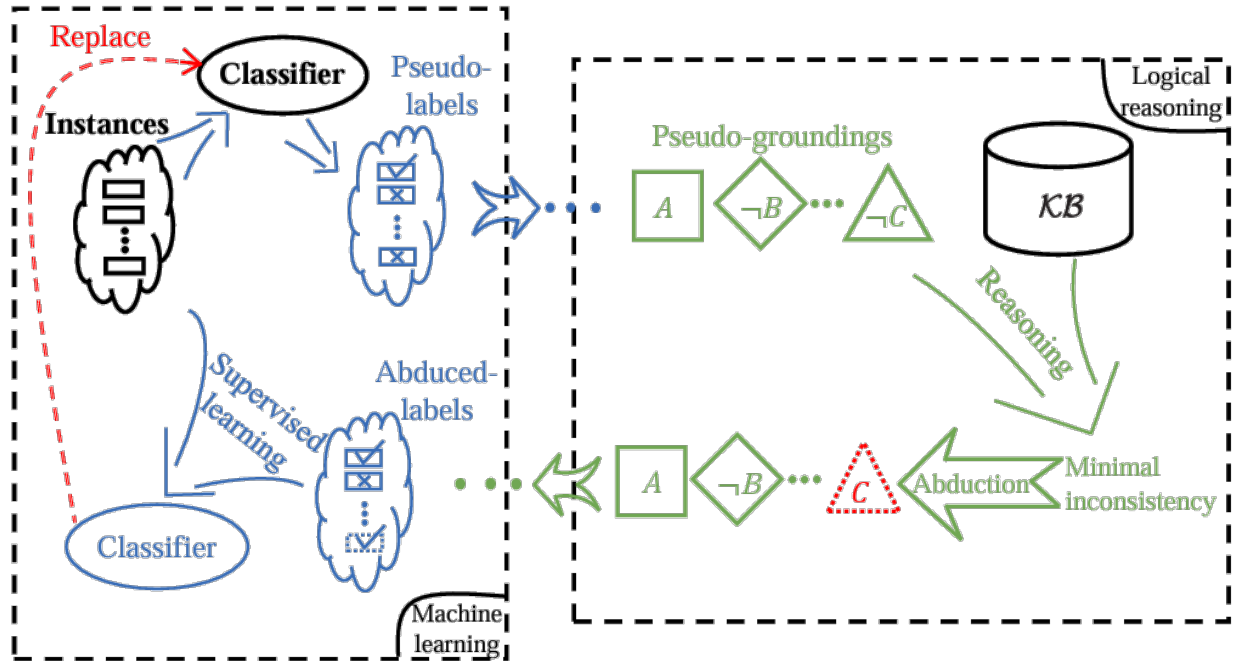
In **Abductive Learning (ABL)**, we assume that, in addition to data, there is also a knowledge base \mathcal{KB} containing domain knowledge at our disposal. We aim for the classifier f to make correct predictions on data instances $\{x_1, \dots, x_m\}$, and meanwhile, the pseudo-groundings grounded by the prediction $\{f(x_1), \dots, f(x_m)\}$ should be compatible with \mathcal{KB} .

The process of ABL is as follows:

1. Upon receiving data instances $\{x_1, \dots, x_m\}$ as input, pseudo-labels $\{f(x_1), \dots, f(x_m)\}$ are predicted by a data-driven classifier model.
2. These pseudo-labels are then converted into pseudo-groundings \mathcal{O} that are acceptable for logical reasoning.
3. Conduct joint reasoning with \mathcal{KB} to find any inconsistencies. If found, the pseudo-groundings that lead to minimal inconsistency can be identified.
4. Modify the identified facts through **abductive reasoning** (or, **abduction**), returning revised pseudo-groundings $\Delta(\mathcal{O})$ which are compatible with \mathcal{KB} .
5. These revised pseudo-groundings are converted back to the form of pseudo-labels, and used like ground-truth labels in conventional supervised learning to train a new classifier.
6. The new classifier will then be adopted to replace the previous one in the next iteration.

This above process repeats until the classifier is no longer updated, or the pseudo-groundings \mathcal{O} are compatible with the knowledge base.

The following figure illustrates this process:



We can observe that in the above figure, the left half involves machine learning, while the right half involves logical reasoning. Thus, the entire Abductive Learning process is a continuous cycle of machine learning and logical reasoning. This effectively forms a paradigm that is dual-driven by both data and domain knowledge, integrating and balancing the use of machine learning and logical reasoning in a unified model.

For more information about ABL, please refer to [References](#).

What is Abductive Reasoning?

Abductive reasoning, also known as abduction, refers to the process of selectively inferring certain facts and hypotheses that explain phenomena and observations based on background knowledge. Unlike deductive reasoning, which leads to definitive conclusions, abductive reasoning may arrive at conclusions that are plausible but not conclusively proven.

In ABL, given \mathcal{KB} (typically expressed in first-order logic clauses), one can perform both deductive and abductive reasoning. Deductive reasoning allows deriving b from a , while abductive reasoning allows inferring a as an explanation of b . In other words, deductive reasoning and abductive reasoning differ in which end, right or left, of the proposition " $a \models b$ " serves as conclusion.

2.2 Installation

2.2.1 Install from PyPI

The easiest way to install ABLkit is using `pip`:

```
pip install ablkit
```

2.2.2 Install from Source

Alternatively, to install from source code, sequentially run following commands in your terminal/command line.

```
git clone https://github.com/AbductiveLearning/ABLkit.git
cd ABLkit
pip install -v -e .
```

2.2.3 (Optional) Install SWI-Prolog

If the use of a *Prolog-based knowledge base* is necessary, the installation of *SWI-Prolog* is also required:

For Linux users:

```
sudo apt-get install swi-prolog
```

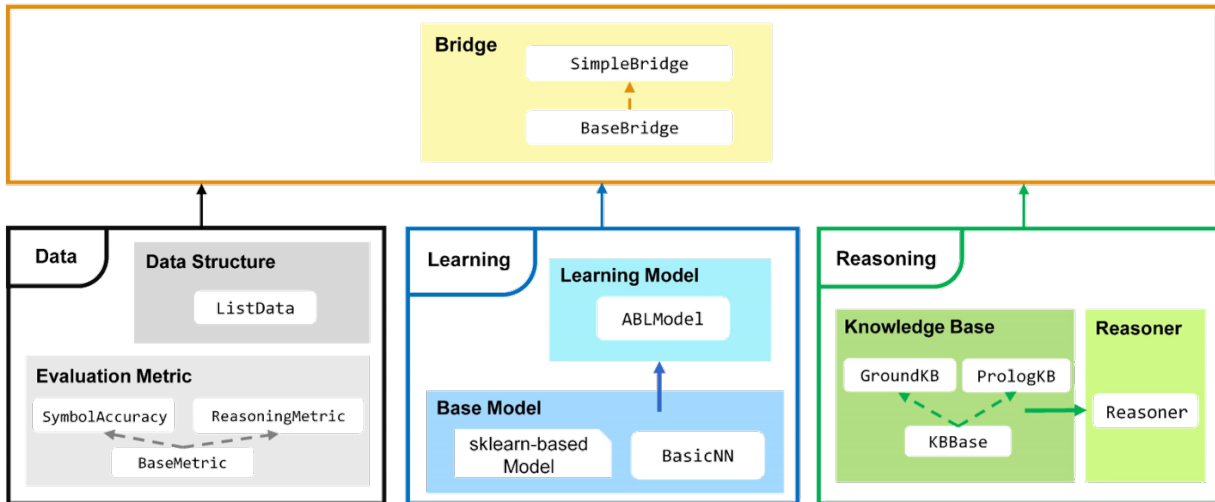
For Windows and Mac users, please refer to the [SWI-Prolog Install Guide](#).

Learn the Basics || [Quick Start](#) || [Dataset & Data Structure](#) || [Learning Part](#) || [Reasoning Part](#) || [Evaluation Metrics](#) || [Bridge](#)

2.3 Learn the Basics

2.3.1 Modules in ABLkit

ABLkit is an efficient toolkit for [Abductive Learning](#) (ABL), a paradigm which integrates machine learning and logical reasoning in a balanced-loop. ABLkit comprises three primary parts: **Data**, **Learning**, and **Reasoning**, corresponding to the three pivotal components of current AI: data, models, and knowledge. Below is an overview of the ABLkit.



Data part efficiently manages data storage, operations, and evaluations. It includes the `ListData` class, which defines the data structures used in ABLkit, and comprises common data operations like insertion, deletion, retrieval, slicing, etc. Additionally, it contains a series of evaluation metrics such as `SymbolAccuracy` and `ReasoningMetric` (both specialized metrics inherited from the `BaseMetric` class), for evaluating performance from a data perspective.

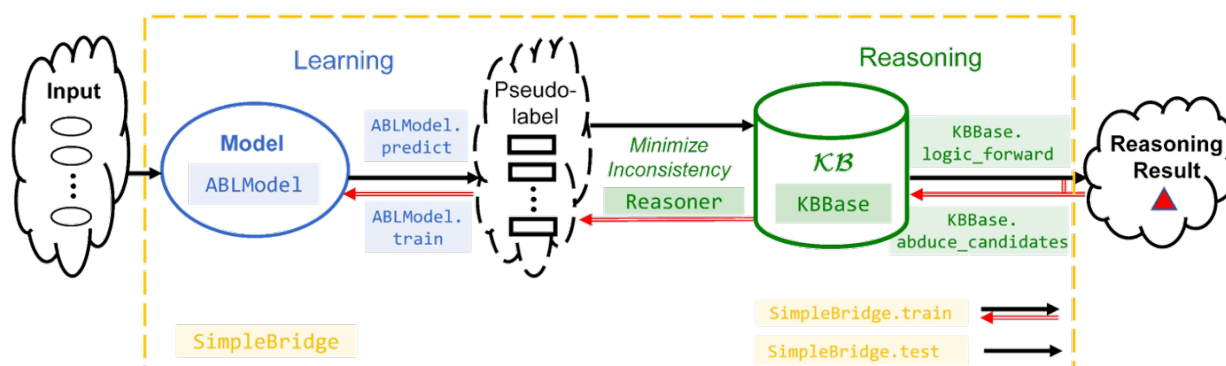
Learning part focuses on creating, training, and utilizing machine learning models. The `ABLModel` class is the central class that encapsulates the machine learning model. This class is compatible with various frameworks, including those based on scikit-learn or PyTorch neural networks constructed by the `BasicNN` class.

Reasoning part concentrates on constructing domain knowledge and performing reasoning. The `KBBase` class allows users to customize a domain knowledge base. For diverse types of knowledge, we also offer implementations like `GroundKB` and `PrologKB` (both inherited from the `KBBase` class). The latter, for instance, imports knowledge bases via Prolog files. Upon building the knowledge base, the `Reasoner` class is responsible for minimizing the inconsistency between the knowledge base and data.

The integration of these three parts is achieved through the Bridge part, which features the `SimpleBridge` class (derived from the `BaseBridge` class). The Bridge part synthesizes data, learning, and reasoning, facilitating the training and testing of the entire ABL framework.

2.3.2 Use ABLkit Step by Step

In a typical ABL process, as illustrated below, data inputs are first predicted by the learning model `ABLModel.predict`, and the outcomes are pseudo-labels. These labels then pass through deductive reasoning of the domain knowledge base `KBBase.logic_forward` to obtain the reasoning result. During training, alongside the aforementioned forward flow (i.e., prediction \rightarrow deduction reasoning), there also exists a reverse flow, which starts from the reasoning result and involves abductive reasoning `KBBase.abduce_candidates` to generate possible revised pseudo-labels. Subsequently, these pseudo-labels are processed to minimize inconsistencies with the learning part. They in turn revise the outcomes of the learning model, which are then fed back for further training `ABLModel.train`.



To implement this process, the following five steps are necessary:

1. Prepare **datasets**

Prepare the data's input, ground truth for pseudo-labels (optional), and ground truth for reasoning results.

2. Build the learning part

Build a machine learning base model that can predict inputs to pseudo-labels. Then, use `ABLModel` to encapsulate the base model.

3. Build the reasoning part

Define a knowledge base by building a subclass of `KBBase`, specifying how to process pseudo-labels to reasoning results. Also, create a `Reasoner` for minimizing inconsistencies between the knowledge base and data.

4. Define evaluation metrics

Define the metrics by building a subclass of `BaseMetric`. The metrics will specify how to measure performance during the training and testing of the ABL framework.

5. Bridge learning and reasoning

Use `SimpleBridge` to bridge the learning and reasoning part for integrated training and testing.

[Learn the Basics](#) || [Quick Start](#) || [Dataset & Data Structure](#) || [Learning Part](#) || [Reasoning Part](#) || [Evaluation Metrics](#) || [Bridge](#)

2.4 Quick Start

We use the MNIST Addition task as a quick start example. In this task, pairs of MNIST handwritten images and their sums are given, along with a domain knowledge base which contains information on how to perform addition operations. Our objective is to input a pair of handwritten images and accurately determine their sum. Refer to the links in each section to dive deeper.

2.4.1 Working with Data

ABLkit requires data in the format of $(X, \text{gt_pseudo_label}, Y)$ where X is a list of input examples containing instances, `gt_pseudo_label` is the ground-truth label of each example in X and Y is the ground-truth reasoning result of each example in X . Note that `gt_pseudo_label` is only used to evaluate the machine learning model's performance but not to train it.

In the MNIST Addition task, the data loading looks like

```
# The 'datasets' module below is located in 'examples/mnist_add/'
from datasets import get_dataset

# train_data and test_data are tuples in the format of (X, gt_pseudo_label, Y)
train_data = get_dataset(train=True)
test_data = get_dataset(train=False)
```

Read more about [preparing datasets](#).

2.4.2 Building the Learning Part

Learning part is constructed by first defining a base model for machine learning. ABLkit offers considerable flexibility, supporting any base model that conforms to the scikit-learn style (which requires the implementation of `fit` and `predict` methods), or a PyTorch-based neural network (which has defined the architecture and implemented `forward` method). In this example, we build a simple LeNet5 network as the base model.

```
# The 'models' module below is located in 'examples/mnist_add/'
from models.nn import LeNet5

cls = LeNet5(num_classes=10)
```

To facilitate uniform processing, ABLkit provides the `BasicNN` class to convert a PyTorch-based neural network into a format compatible with scikit-learn models. To construct a `BasicNN` instance, aside from the network itself, we also need to define a loss function, an optimizer, and the computing device.

```
import torch
from ablkit.learning import BasicNN

loss_fn = torch.nn.CrossEntropyLoss()
```

(continues on next page)

(continued from previous page)

```
optimizer = torch.optim.RMSprop(cls.parameters(), lr=0.001)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
base_model = BasicNN(model=cls, loss_fn=loss_fn, optimizer=optimizer, device=device)
```

The base model built above is trained to make predictions on instance-level data (e.g., a single image), while ABL deals with example-level data. To bridge this gap, we wrap the `base_model` into an instance of `ABLModel`. This class serves as a unified wrapper for base models, facilitating the learning part to train, test, and predict on example-level data, (e.g., images that comprise an equation).

```
from ablkit.learning import ABLModel

model = ABLModel(base_model)
```

Read more about [building the learning part](#).

2.4.3 Building the Reasoning Part

To build the reasoning part, we first define a knowledge base by creating a subclass of `KBBBase`. In the subclass, we initialize the `pseudo_label_list` parameter and override the `logic_forward` method, which specifies how to perform (deductive) reasoning that processes pseudo-labels of an example to the corresponding reasoning result. Specifically, for the MNIST Addition task, this `logic_forward` method is tailored to execute the sum operation.

```
from ablkit.reasoning import KBBBase

class AddKB(KBBBase):
    def __init__(self, pseudo_label_list=list(range(10))):
        super().__init__(pseudo_label_list)

    def logic_forward(self, nums):
        return sum(nums)

kb = AddKB()
```

Next, we create a reasoner by instantiating the class `Reasoner`, passing the knowledge base as a parameter. Due to the indeterminism of abductive reasoning, there could be multiple candidate pseudo-labels compatible with the knowledge base. In such scenarios, the reasoner can minimize inconsistency and return the pseudo-label with the highest consistency.

```
from ablkit.reasoning import Reasoner

reasoner = Reasoner(kb)
```

Read more about [building the reasoning part](#).

2.4.4 Building Evaluation Metrics

ABLkit provides two basic metrics, namely `SymbolAccuracy` and `ReasoningMetric`, which are used to evaluate the accuracy of the machine learning model's predictions and the accuracy of the `logic_forward` results, respectively.

```
from ablkit.data.evaluation import ReasoningMetric, SymbolAccuracy

metric_list = [SymbolAccuracy(), ReasoningMetric(kb=kb)]
```

Read more about [building evaluation metrics](#)

2.4.5 Bridging Learning and Reasoning

Now, we use `SimpleBridge` to combine learning and reasoning in a unified ABL framework.

```
from ablkit.bridge import SimpleBridge

bridge = SimpleBridge(model, reasoner, metric_list)
```

Finally, we proceed with training and testing.

```
bridge.train(train_data, loops=1, segment_size=0.01)
bridge.test(test_data)
```

Read more about [bridging machine learning and reasoning](#).

[Learn the Basics](#) || [Quick Start](#) || **[Dataset & Data Structure](#)** || [Learning Part](#) || [Reasoning Part](#) || [Evaluation Metrics](#) || [Bridge](#)

2.5 Dataset & Data Structure

In this section, we will look at the dataset and data structure in ABLkit.

```
import torch
from ablkit.data.structures import ListData
```

2.5.1 Dataset

ABLkit requires user data to be either structured as a tuple (`X`, `gt_pseudo_label`, `Y`) or a `ListData` (the underlying data structure utilized in ABLkit, cf. the next section) object with `X`, `gt_pseudo_label` and `Y` attributes. Regardless of the chosen format, the data should encompass three essential components:

- `X`: `List[List[Any]]`
A list of sublists representing the input data. We refer to each sublist in `X` as an **example** and each example may contain several **instances**.
- `gt_pseudo_label`: `List[List[Any]]`, optional
A list of sublists with each sublist representing ground-truth pseudo-labels of an example. Each pseudo-label in the sublist serves as ground-truth for each **instance** within the example.





Note: `gt_pseudo_label` is only used to evaluate the performance of the learning part but not to train the model. If the pseudo-label of the instances in the datasets are unlabeled, `gt_pseudo_label` should be `None`.




- `Y: List[Any]`

A list representing the ground-truth reasoning result for each **example** in `X`.

Warning: The length of `X`, `gt_pseudo_label` (if not `None`) and `Y` should be the same. Also, each sublist in `gt_pseudo_label` should have the same length as the sublist in `X`.

As an illustration, in the MNIST Addition task, the data are organized as follows:

```
X: [[, , ..., , ]
gt_pseudo_label: [[3, 5], [4, 9], ..., [2, 0]]
Y: [ 8 , 13 , ..., 2 ]
```

where each sublist in `X`, e.g., [, ], is a data example and each image in the sublist, e.g., , is an instance.

2.5.2 Data Structure

Besides the user-provided dataset, various forms of data are utilized and dynamically generated throughout the training and testing process of ABL framework. Examples include raw data, predicted pseudo-label, abduced pseudo-label, pseudo-label indices, etc. To manage this diversity and ensure a stable, versatile interface, ABLkit employs [abstract data interfaces](#) to encapsulate different forms of data that will be used in the total learning process.

`ListData` is the underlying abstract data interface utilized in ABLkit. As the fundamental data structure, `ListData` implements commonly used data manipulation methods and is responsible for transferring data between various components of ABL, ensuring that stages such as prediction, abductive reasoning, and training can utilize `ListData` as a unified input format. Before proceeding to other stages, user-provided datasets will be firstly converted into `ListData`.

Besides providing a tuple of (`X`, `gt_pseudo_label`, `Y`), ABLkit also allows users to directly supply data in `ListData` format, which similarly requires the inclusion of these three attributes. The following code shows the basic usage of `ListData`. More information can be found in the [API documentation](#).

```
# Prepare data
X = [list(torch.randn(3, 28, 28)), list(torch.randn(3, 28, 28))]
gt_pseudo_label = [[1, 2, 3], [4, 5, 6]]
Y = [1, 2]

# Convert data into ListData
data = ListData(X=X, Y=Y, gt_pseudo_label=gt_pseudo_label)

# Get data
X = data.X
Y = data.Y
gt_pseudo_label = data.gt_pseudo_label

# Set data
```

(continues on next page)

(continued from previous page)

```
data.X = X
data.Y = Y
data.gt_pseudo_label = gt_pseudo_label
```

[Learn the Basics](#) || [Quick Start](#) || [Dataset & Data Structure](#) || **Learning Part** || [Reasoning Part](#) || [Evaluation Metrics](#) || [Bridge](#)

2.6 Learning Part

In this section, we will look at how to build the learning part.

In ABLkit, building the learning part involves two steps:

1. Build a machine learning base model used to make predictions on instance-level data.
2. Instantiate an ABLModel with the base model, which enables the learning part to process example-level data.

```
import sklearn
import torchvision
from ablkit.learning import BasicNN, ABLModel
```

2.6.1 Building a base model

ABL toolkit allows the base model to be one of the following forms:

1. Any machine learning model conforming to the scikit-learn style, i.e., models which has implemented the `fit` and `predict` methods;
2. A PyTorch-based neural network, provided it has defined the architecture and implemented the `forward` method.

For a scikit-learn model, we can directly use the model itself as a base model. For example, we can customize our base model by a KNN classifier:

```
base_model = sklearn.neighbors.KNeighborsClassifier(n_neighbors=3)
```

For a PyTorch-based neural network, we need to encapsulate it within a BasicNN object to create a base model. For example, we can customize our base model by a pre-trained ResNet-18:

```
# Load a PyTorch-based neural network
cls = torchvision.models.resnet18(pretrained=True)

# loss function and optimizer are used for training
loss_fn = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cls.parameters())

base_model = BasicNN(cls, loss_fn, optimizer)
```

BasicNN

BasicNN is a wrapper class for PyTorch-based neural networks, which enables them to work as scikit-learn models. It encapsulates the neural network, loss function, optimizer, and other elements into a single object, which can be used as a base model.

Besides the necessary methods required to instantiate an `ABLModel`, i.e., `fit` and `predict`, `BasicNN` also implements the following methods:

Method	Function
<code>train_epoch(data_loader)</code>	Train the neural network for one epoch.
<code>predict_proba(X)</code>	Predict the class probabilities of <code>X</code> .
<code>score(X, y)</code>	Calculate the accuracy of the model on test data.
<code>save(epoch_id, save_path)</code>	Save the model.
<code>load(load_path)</code>	Load the model.

2.6.2 Instantiating an ABLModel

Typically, base model is trained to make predictions on instance-level data, and can not directly process example-level data, which is not suitable for most neural-symbolic tasks. ABLkit provides the `ABLModel` to solve this problem. This class serves as a unified wrapper for all base models, which enables the learning part to train, test, and predict on example-level data.

Generally, we can simply instantiate an `ABLModel` by:

```
# Instantiate an ABLModel
model = ABLModel(base_model)
```

[Learn the Basics](#) || [Quick Start](#) || [Dataset & Data Structure](#) || [Learning Part](#) || **Reasoning Part** || [Evaluation Metrics](#) || [Bridge](#)

2.7 Reasoning part

In this section, we will look at how to build the reasoning part, which leverages domain knowledge and performs deductive or abductive reasoning. In ABLkit, building the reasoning part involves two steps:

1. Build a knowledge base by creating a subclass of `KBBase`, which specifies how to process pseudo-label of an example to the reasoning result.
2. Create a reasoner by instantiating the class `Reasoner` to minimize inconsistencies between the knowledge base and pseudo labels predicted by the learning part.

```
from ablkit.reasoning import KBBase, GroundKB, PrologKB, Reasoner
```

2.7.1 Building a knowledge base

Generally, we can create a subclass derived from `KBase` to build our own knowledge base. In addition, ABLkit also offers several predefined subclasses of `KBase` (e.g., `PrologKB` and `GroundKB`), which we can utilize to build our knowledge base more conveniently.

Building a knowledge base from `KBase`

For the user-built KB from `KBase` (a derived subclass), it's only required to pass the `pseudo_label_list` parameter in the `__init__` method and override the `logic_forward` method:

- `pseudo_label_list` is the list of possible pseudo-labels (also, the output of the machine learning model).
- `logic_forward` defines how to perform (deductive) reasoning, i.e. matching each example's pseudo-labels to its reasoning result.

Note: Generally, the overridden method `logic_forward` provided by the user accepts only one parameter, `pseudo_label` (pseudo-labels of an example). However, for certain scenarios, deductive reasoning in the knowledge base may necessitate information from the input. In these scenarios, `logic_forward` can also accept two parameters: `pseudo_label` and `x`. See examples in [Zoo](#).

After that, other operations, including how to perform abductive reasoning, will be **automatically** set up.

MNIST Addition example

As an example, the `pseudo_label_list` passed in MNIST Addition is all the possible digits, namely, `[0, 1, 2, ..., 9]`, and the `logic_forward` should be: "Add the two pseudo-labels to get the result.". Therefore, the construction of the KB (`add_kb`) for MNIST Addition would be:

```
class AddKB(KBase):
    def __init__(self, pseudo_label_list=list(range(10))):
        super().__init__(pseudo_label_list)

    def logic_forward(self, pseudo_labels):
        return sum(pseudo_labels)

add_kb = AddKB()
```

and (deductive) reasoning in `add_kb` would be:

```
pseudo_labels = [1, 2]
reasoning_result = add_kb.logic_forward(pseudo_labels)
print(f"Reasoning result of pseudo-labels {pseudo_labels} is {reasoning_result}.")
```

Out:

```
Reasoning result of pseudo-labels [1, 2] is 3
```

Other optional parameters

We can also pass the following parameters in the `__init__` method when building our knowledge base:

- `max_err` (float, optional), specifying the upper tolerance limit when comparing the similarity between the reasoning result of pseudo-labels and the ground truth during abductive reasoning. This is only applicable when the reasoning result is of a numerical type. This is particularly relevant for regression problems where exact matches might not be feasible. Defaults to `1e-10`. See [an example](#).
- `use_cache` (bool, optional), indicating whether to use cache to store previous candidates (pseudo-labels generated from abductive reasoning) to speed up subsequent abductive reasoning operations. Defaults to `True`. For more information of abductive reasoning, please refer to [this](#).
- `cache_size` (int, optional), specifying the maximum cache size. This is only operational when `use_cache` is set to `True`. Defaults to `4096`.

Building a knowledge base from Prolog file

When aiming to leverage knowledge base from an external Prolog file (which contains how to perform reasoning), we can directly create an instance of class `PrologKB`. Upon instantiation of `PrologKB`, we are required to pass the `pseudo_label_list` (same as `KBBase`) and `pl_file` (the Prolog file) in the `__init__` method.

What is a Prolog file?

A Prolog file (typically have the extension `.pl`) is a script or source code file written in the Prolog language. Prolog is a logic programming language where the logic is represented as facts (basic assertions about some world) and rules (logical statements that describe the relationships between facts). A computation is initiated by running a query over these facts and rules. See some Prolog examples in [SWISH](#).

After the instantiation, other operations, including how to perform abductive reasoning, will also be **automatically** set up.

Warning: Note that to use the default logic forward and abductive reasoning methods in this class, the Prolog (`.pl`) file should contain a rule with a strict format: `logic_forward(Pseudo_labels, Res)`. Otherwise, we might have to override `logic_forward` and `get_query_string` to allow for more adaptable usage.

MNIST Addition example (cont.)

As an example, we can first write a Prolog file for the MNIST Addition example as the following code, and then save it as `add.pl`.

```
pseudo_label(N) :- between(0, 9, N).
logic_forward([Z1, Z2], Res) :- pseudo_label(Z1), pseudo_label(Z2), Res is Z1+Z2.
```

Afterwards, the construction of knowledge base from Prolog file (`add_prolog_kb`) would be as follows:

```
add_prolog_kb = PrologKB(pseudo_label_list=list(range(10)), pl_file="add.pl")
```

Building a knowledge base with GKB from GroundKB

We can also inherit from class GroundKB to build our own knowledge base. In this way, the knowledge built will have a Ground KB (GKB).

What is Ground KB?

Ground KB is a knowledge base prebuilt upon class initialization, storing all potential candidates along with their respective reasoning result. The key advantage of having a Ground KB is that it may accelerate abductive reasoning.

GroundKB is a subclass of GKBBase. Similar to KBBase, we are required to pass the `pseudo_label_list` parameter in the `__init__` method and override the `logic_forward` method, and are allowed to pass other *optional parameters*. Additionally, we are required pass the `GKB_len_list` parameter in the `__init__` method.

- `GKB_len_list` is the list of possible lengths for pseudo-labels of an example.

After that, other operations, including auto-construction of GKB, and how to perform abductive reasoning, will be **automatically** set up.

MNIST Addition example (cont.)

As an example, the `GKB_len_list` for MNIST Addition should be `[2]`, since all pseudo-labels in the example consist of two digits. Therefore, the construction of KB with GKB (`add_ground_kb`) of MNIST Addition would be as follows. As mentioned, the difference between this and the previously built `add_kb` lies only in the base class from which it is derived and whether an extra parameter `GKB_len_list` is passed.

```
class AddGroundKB(GroundKB):
    def __init__(self, pseudo_label_list=list(range(10)),
                 GKB_len_list=[2]):
        super().__init__(pseudo_label_list, GKB_len_list)

    def logic_forward(self, nums):
        return sum(nums)

add_ground_kb = AddGroundKB()
```

Performing abductive reasoning in the knowledge base

As mentioned in *What is Abductive Reasoning?*, abductive reasoning enables the inference of candidates (i.e., possible pseudo-labels) as potential explanations for the reasoning result. Also, in Abductive Learning where an observation (pseudo-labels of an example predicted by the learning part) is available, we aim to let the candidate do not largely revise the previously identified pseudo-labels.

KBBase (also, GroundKB and PrologKB) implement the method `abduce_candidates(pseudo_label, y, x, max_revision_num, require_more_revision)` for performing abductive reasoning, where the parameters are:

- `pseudo_label`, pseudo-labels of an example, usually generated by the learning part. They are to be revised by abductive reasoning.
- `y`, the ground truth of the reasoning result for the example. The returned candidates should be compatible with it.
- **`x`, the corresponding input example. If the information from the input**
is not required in the reasoning process, then this parameter will not have any effect.

- `max_revision_num`, an int value specifying the upper limit on the number of revised labels for each example.
- `require_more_revision`, an int value specifying additional number of revisions permitted beyond the minimum required. (e.g., If we set it to 0, even if `max_revision_num` is set to a high value, the method will only output candidates with the minimum possible revisions.)

And it returns a list of candidates (i.e., revised pseudo-labels of the example) that are all compatible with `y`.

MNIST Addition example (cont.)

As an example, with MNIST Addition, the candidates returned by `add_kb.abduce_candidates` would be as follows:

pseudo_label	y	max_revision_num	require_more_address	Output
[1,1]	8	1	0	[[1,7], [7,1]]
[1,1]	8	1	1	[[1,7], [7,1]]
[1,1]	8	2	0	[[1,7], [7,1]]
[1,1]	8	2	1	[[1,7], [7,1], [2,6], [6,2], [3,5], [5,3], [4,4]]
[1,1]	11	1	0	[]

As another example, if we set the `max_err` of `AddKB` to be 1 instead of the default 1e-10, the tolerance limit for consistency will be higher, hence the candidates returned would be:

pseudo_label	y	max_revision_num	require_more_address	Output
[1,1]	8	1	0	[[1,7], [7,1], [1,6], [6,1], [1,8], [8,1]]
[1,1]	11	1	0	[[1,9], [9,1]]

2.7.2 Creating a reasoner

After building our knowledge base, the next step is creating a reasoner. Due to the indeterminism of abductive reasoning, there could be multiple candidates compatible with the knowledge base. When this happens, reasoner can minimize inconsistencies between the knowledge base and pseudo-labels predicted by the learning part, and then return **only one** candidate that has the highest consistency.

We can create a reasoner simply by instantiating class `Reasoner` and passing our knowledge base as a parameter. As an example for MNIST Addition, the reasoner definition would be:

```
reasoner_add = Reasoner(kb_add)
```

When instantiating, besides the required knowledge base, we may also specify:

- `max_revision` (int or float, optional), specifies the upper limit on the number of revisions for each example when performing *abductive reasoning in the knowledge base*. If float, denotes the fraction of the total length that can be revised. A value of -1 implies no restriction on the number of revisions. Defaults to -1.
- `require_more_revision` (int, optional), Specifies additional number of revisions permitted beyond the minimum required when performing *abductive reasoning in the knowledge base*. Defaults to 0.
- `use_zoopt` (bool, optional), indicating whether to use the `ZOOpt library`, which is a library for zeroth-order optimization that can be used to accelerate consistency minimization. Defaults to False.
- `dist_func` (str, optional), specifying the distance function to be used when determining consistency between your prediction and candidate returned from knowledge base. This can be either a user-defined function or one that is predefined. Valid predefined options include “hamming”, “confidence” and “avg_confidence”. For

“hamming”, it directly calculates the Hamming distance between the predicted pseudo-label in the data example and candidate. For “confidence”, it calculates the confidence distance between the predicted probabilities in the data example and each candidate, where the confidence distance is defined as $1 - \text{the product of prediction probabilities in "confidence"}$ and $1 - \text{the average of prediction probabilities in "avg_confidence"}$. Defaults to “confidence”.

- **idx_to_label (dict, optional), a mapping from index in the base model to label.**

If not provided, a default order-based index to label mapping is created. Defaults to None.

The main method implemented by Reasoner is `abduce(data_example)`, which obtains the most consistent candidate based on the distance function defined in `dist_func`.

MNIST Addition example (cont.)

As an example, consider these data examples for MNIST Addition:

```
# favor "1" for the first label
prob1 = [[0, 0.99, 0, 0, 0, 0, 0.01, 0, 0],
          [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]]

# favor "7" for the first label
prob2 = [[0, 0.01, 0, 0, 0, 0, 0.99, 0, 0],
          [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]]

example1 = ListData()
example1.pred_pseudo_label = [1, 1]
example1.pred_prob = prob1
example1.Y = 8

example2 = ListData()
example2.pred_pseudo_label = [1, 1]
example2.pred_prob = prob2
example2.Y = 8
```

The compatible candidates after abductive reasoning for both examples would be `[[1,7], [7,1]]`. However, when the reasoner calls `abduce` to select only one candidate based on the “confidence” distance function, the output would differ for each example:

```
reasoner_add = Reasoner(kb_add, dist_func="confidence")
candidate1 = reasoner_add.abduce(example1)
candidate2 = reasoner_add.abduce(example2)
print(f"The outputs for example1 and example2 are {candidate1} and {candidate2},\n↪respectively.")
```

Out:

```
The outputs for example1 and example2 are [1,7] and [7,1], respectively.
```

Specifically, as mentioned before, “confidence” calculates the distance between the data example and candidates based on the confidence derived from the predicted probability. Take `example1` as an example, the `pred_prob` in it indicates a higher confidence that the first label should be “1” rather than “7”. Therefore, among the candidates `[1,7]` and `[7,1]`, it would be closer to `[1,7]` (as its first label is “1”).

[Learn the Basics](#) || [Quick Start](#) || [Dataset & Data Structure](#) || [Learning Part](#) || [Reasoning Part](#) || [Evaluation Metrics](#) || [Bridge](#)

2.8 Evaluation Metrics

In this section, we will look at how to build evaluation metrics.

```
from ablkit.data.evaluation import BaseMetric, SymbolAccuracy, ReasoningMetric
```

ABLkit separates the evaluation process from model training and testing as an independent class, `BaseMetric`. The training and testing processes are implemented in the `BaseBridge` class, so metrics are used by this class and its subclasses. After building a bridge with a list of `BaseMetric` instances, these metrics will be used by the `bridge.valid` method to evaluate the model performance during training and testing.

To customize our own metrics, we need to inherit from `BaseMetric` and implement the `process` and `compute_metrics` methods.

- The `process` method accepts a batch of model prediction and saves the information to `self.results` property after processing this batch.
- The `compute_metrics` method uses all the information saved in `self.results` to calculate and return a dict that holds the evaluation results.

Besides, we can assign a `str` to the `prefix` argument of the `__init__` function. This string is automatically prefixed to the output metric names. For example, if we set `prefix="mnist_add"`, the output metric name will be `character_accuracy`. We provide two basic metrics, namely `SymbolAccuracy` and `ReasoningMetric`, which are used to evaluate the accuracy of the machine learning model's predictions and the accuracy of the final reasoning results, respectively. Using `SymbolAccuracy` as an example, the following code shows how to implement a custom metric.

```
class SymbolAccuracy(BaseMetric):
    def __init__(self, prefix: Optional[str] = None) -> None:
        # prefix is used to distinguish different metrics
        super().__init__(prefix)

    def process(self, data_examples: Sequence[dict]) -> None:
        # pred_pseudo_label and gt_pseudo_label are both of type List[List[Any]]
        # and have the same length
        pred_pseudo_label = data_examples.pred_pseudo_label
        gt_pseudo_label = data_examples.gt_pseudo_label

        for pred_z, z in zip(pred_pseudo_label, gt_pseudo_label):
            correct_num = 0
            for pred_symbol, symbol in zip(pred_z, z):
                if pred_symbol == symbol:
                    correct_num += 1
            self.results.append(correct_num / len(z))

    def compute_metrics(self, results: list) -> dict:
        metrics = dict()
        metrics["character_accuracy"] = sum(results) / len(results)
        return metrics
```

Learn the Basics || Quick Start || Dataset & Data Structure || Learning Part || Reasoning Part || Evaluation Metrics || Bridge

2.9 Bridge

In this section, we will look at how to bridge learning and reasoning parts to train the model, which is the fundamental idea of Abductive Learning. ABLkit implements a set of bridge classes to achieve this.

```
from ablkit.bridge import BaseBridge, SimpleBridge
```

BaseBridge is an abstract class with the following initialization parameters:

- `model` is an object of type `ABLModel`. The learning part is wrapped in this object.
- `reasoner` is an object of type `Reasoner`. The reasoning part is wrapped in this object.

BaseBridge has the following important methods that need to be overridden in subclasses:

Method Signature	Description
<code>predict(data_examples)</code>	Predicts class probabilities and indices for the given data examples.
<code>abduce_pseudo_label(data_examples)</code>	Abduces pseudo-labels for the given data examples.
<code>idx_to_pseudo_label(data_examples)</code>	Converts indices to pseudo-labels using the provided or default mapping.
<code>pseudo_label_to_idx(data_examples)</code>	Converts pseudo-labels to indices using the provided or default remapping.
<code>train(train_data)</code>	Train the model.
<code>test(test_data)</code>	Test the model.

where `train_data` and `test_data` are both in the form of a tuple or a `ListData`. Regardless of the form, they all need to include three components: `X`, `gt_pseudo_label` and `Y`. Since `ListData` is the underlying data structure used throughout the ABLkit, tuple-formed data will be firstly transformed into `ListData` in the `train` and `test` methods, and such `ListData` instances are referred to as `data_examples`. More details can be found in [preparing datasets](#).

`SimpleBridge` inherits from `BaseBridge` and provides a basic implementation. Besides the `model` and `reasoner`, `SimpleBridge` has an extra initialization argument, `metric_list`, which will be used to evaluate model performance. Its training process involves several Abductive Learning loops and each loop consists of the following five steps:

1. Predict class probabilities and indices for the given data examples.
2. Transform indices into pseudo-labels.
3. Revise pseudo-labels based on abductive reasoning.
4. Transform the revised pseudo-labels to indices.
5. Train the model.

The fundamental part of the `train` method is as follows:

```
def train(self, train_data, loops=50, segment_size=10000):
    """
    Parameters
    -----
    train_data : Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]],
    ↳List[Any]]
        Training data should be in the form of ``(X, gt_pseudo_label, Y)`` or a ``ListData``
        object with ``X``, ``gt_pseudo_label`` and ``Y`` attributes.
        - ``X`` is a list of sublists representing the input data.
        - ``gt_pseudo_label`` is only used to evaluate the performance of the ``ABLModel``.
    ↳but not
```

(continues on next page)

(continued from previous page)

```

        to train. ``gt_pseudo_label`` can be ``None``.
        - ``Y`` is a list representing the ground truth reasoning result for each sublist.
    ↪in ``X``.
        loops : int
            Learning part and Reasoning part will be iteratively optimized for ``loops``
    ↪times.
        segment_size : Union[int, float]
            Data will be split into segments of this size and data in each segment
            will be used together to train the model.
        """
        if isinstance(train_data, ListData):
            data_examples = train_data
        else:
            data_examples = self.data_preprocess(*train_data)

        if isinstance(segment_size, float):
            segment_size = int(segment_size * len(data_examples))

        for loop in range(loops):
            for seg_idx in range((len(data_examples) - 1) // segment_size + 1):
                sub_data_examples = data_examples[
                    seg_idx * segment_size : (seg_idx + 1) * segment_size
                ]
                self.predict(sub_data_examples)           # 1
                self.idx_to_pseudo_label(sub_data_examples) # 2
                self.abduce_pseudo_label(sub_data_examples) # 3
                self.pseudo_label_to_idx(sub_data_examples) # 4
                loss = self.model.train(sub_data_examples) # 5, self.model is an
    ↪ABLModel object

```

2.10 MNIST Addition

Below shows an implementation of **MNIST Addition**. In this task, pairs of MNIST handwritten images and their sums are given, along with a domain knowledge base containing information on how to perform addition operations. The task is to recognize the digits of handwritten images and accurately determine their sum.

Intuitively, we first use a machine learning model (learning part) to convert the input images to digits (we call them pseudo-labels), and then use the knowledge base (reasoning part) to calculate the sum of these digits. Since we do not have ground-truth of the digits, in Abductive Learning, the reasoning part will leverage domain knowledge and revise the initial digits yielded by the learning part through abductive reasoning. This process enables us to further update the machine learning model.

```

# Import necessary libraries and modules
import os.path as osp

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from torch.optim import RMSprop, lr_scheduler

```

(continues on next page)

(continued from previous page)

```

from ablkit.bridge import SimpleBridge
from ablkit.data.evaluation import ReasoningMetric, SymbolAccuracy
from ablkit.learning import ABLModel, BasicNN
from ablkit.reasoning import KBBase, Reasoner
from ablkit.utils import ABLLogger, print_log

from datasets import get_dataset
from models.nn import LeNet5

```

2.10.1 Working with Data

First, we get the training and testing datasets:

```

train_data = get_dataset(train=True, get_pseudo_label=True)
test_data = get_dataset(train=False, get_pseudo_label=True)

```

`train_data` and `test_data` share identical structures: tuples with three components: `X` (list where each element is a list of two images), `gt_pseudo_label` (list where each element is a list of two digits, i.e., pseudo-labels) and `Y` (list where each element is the sum of the two digits). The length and structures of datasets are illustrated as follows.

Note: `gt_pseudo_label` is only used to evaluate the performance of the learning part but not to train the model.

```

print(f"Both train_data and test_data consist of 3 components: X, gt_pseudo_label, Y")
print("\n")
train_X, train_gt_pseudo_label, train_Y = train_data
print(f"Length of X, gt_pseudo_label, Y in train_data: " +
      f"{len(train_X)}, {len(train_gt_pseudo_label)}, {len(train_Y)}")
test_X, test_gt_pseudo_label, test_Y = test_data
print(f"Length of X, gt_pseudo_label, Y in test_data: " +
      f"{len(test_X)}, {len(test_gt_pseudo_label)}, {len(test_Y)}")
print("\n")

X_0, gt_pseudo_label_0, Y_0 = train_X[0], train_gt_pseudo_label[0], train_Y[0]
print(f"X is a {type(train_X).__name__}, " +
      f"with each element being a {type(X_0).__name__} " +
      f"of {len(X_0)} {type(X_0[0]).__name__}.")
print(f"gt_pseudo_label is a {type(train_gt_pseudo_label).__name__}, " +
      f"with each element being a {type(gt_pseudo_label_0).__name__} " +
      f"of {len(gt_pseudo_label_0)} {type(gt_pseudo_label_0[0]).__name__}.")
print(f"Y is a {type(train_Y).__name__}, " +
      f"with each element being an {type(Y_0).__name__}.")

```

Out:

```

Both train_data and test_data consist of 3 components: X, gt_pseudo_label, Y

Length of X, gt_pseudo_label, Y in train_data: 30000, 30000, 30000
Length of X, gt_pseudo_label, Y in test_data: 5000, 5000, 5000

X is a list, with each element being a list of 2 Tensor.

```

(continues on next page)

(continued from previous page)

```
gt_pseudo_label is a list, with each element being a list of 2 int.
Y is a list, with each element being an int.
```

The *i*th element of *X*, *gt_pseudo_label*, and *Y* together constitute the *i*th data example. As an illustration, in the first data example of the training set, we have:

```
X_0, gt_pseudo_label_0, Y_0 = train_X[0], train_gt_pseudo_label[0], train_Y[0]
print(f"X in the first data example (a list of two images):")
plt.subplot(1,2,1)
plt.axis('off')
plt.imshow(X_0[0].squeeze(), cmap='gray')
plt.subplot(1,2,2)
plt.axis('off')
plt.imshow(X_0[1].squeeze(), cmap='gray')
plt.show()
print(f"gt_pseudo_label in the first data example (a list of two ground truth pseudo-
→labels): {gt_pseudo_label_0}")
print(f"Y in the first data example (their sum result): {Y_0}")
```

Out:

```
X in the first data example (a list of two images):
```



```
gt_pseudo_label in the first data example (a list of two ground truth pseudo-
→labels): [7, 5]
Y in the first data example (their sum result): 12
```

2.10.2 Building the Learning Part

To build the learning part, we need to first build a machine learning base model. We use a simple [LeNet-5 neural network](#), and encapsulate it within a BasicNN object to create the base model. BasicNN is a class that encapsulates a PyTorch model, transforming it into a base model with a sklearn-style interface.

```
cls = LeNet5(num_classes=10)
loss_fn = nn.CrossEntropyLoss(label_smoothing=0.1)
optimizer = RMSprop(cls.parameters(), lr=0.001, alpha=0.9)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
scheduler = lr_scheduler.OneCycleLR(optimizer, max_lr=0.001, pct_start=0.1, total_
→steps=100)

base_model = BasicNN(
    cls,
    loss_fn,
    optimizer,
```

(continues on next page)

(continued from previous page)

```

    scheduler=scheduler,
    device=device,
    batch_size=32,
    num_epochs=1,
)

```

BasicNN offers methods like `predict` and `predict_proba`, which are used to predict the class index and the probabilities of each class for images. As shown below:

```

data_instances = [torch.randn(1, 28, 28) for _ in range(32)]
pred_idx = base_model.predict(X=data_instances)
print(f"Predicted class index for a batch of 32 instances: np.ndarray with shape {pred_idx.shape}")
pred_prob = base_model.predict_proba(X=data_instances)
print(f"Predicted class probabilities for a batch of 32 instances: np.ndarray with shape {pred_prob.shape}")

```

Out:

```

Predicted class index for a batch of 32 instances: np.ndarray with shape (32,)
Predicted class probabilities for a batch of 32 instances: np.ndarray with shape (32, 10)

```

However, the base model built above deals with instance-level data (i.e., individual images), and can not directly deal with example-level data (i.e., a pair of images). Therefore, we wrap the base model into `ABLModel`, which enables the learning part to train, test, and predict on example-level data.

```
model = ABLModel(base_model)
```

As an illustration, consider this example of training on example-level data using the `predict` method in `ABLModel`. In this process, the method accepts data examples as input and outputs the class labels and the probabilities of each class for all instances within these data examples.

```

from ablkit.data.structures import ListData
# ListData is a data structure provided by ABLkit that can be used to organize data_
# examples
data_examples = ListData()
# We use the first 100 data examples in the training set as an illustration
data_examples.X = train_X[:100]
data_examples.gt_pseudo_label = train_gt_pseudo_label[:100]
data_examples.Y = train_Y[:100]

# Perform prediction on the 100 data examples
pred_label, pred_prob = model.predict(data_examples)['label'], model.predict(data_
# examples)['prob']
print(f"Predicted class labels for the 100 data examples: \n" +
      f"a list of length {len(pred_label)}, and each element is " +
      f"a {type(pred_label[0]).__name__} of shape {pred_label[0].shape}.\n")
print(f"Predicted class probabilities for the 100 data examples: \n" +
      f"a list of length {len(pred_prob)}, and each element is " +
      f"a {type(pred_prob[0]).__name__} of shape {pred_prob[0].shape}.")

```

Out:

Predicted class labels for the 100 data examples:
a list of length 100, and each element is a ndarray of shape (2,).

Predicted class probabilities for the 100 data examples:
a list of length 100, and each element is a ndarray of shape (2, 10).

2.10.3 Building the Reasoning Part

In the reasoning part, we first build a knowledge base which contains information on how to perform addition operations. We build it by creating a subclass of `KBase`. In the derived subclass, we initialize the `pseudo_label_list` parameter specifying list of possible pseudo-labels, and override the `logic_forward` function defining how to perform (deductive) reasoning.

```
class AddKB(KBase):
    def __init__(self, pseudo_label_list=list(range(10))):
        super().__init__(pseudo_label_list)

    # Implement the deduction function
    def logic_forward(self, nums):
        return sum(nums)

kb = AddKB()
```

The knowledge base can perform logical reasoning (both deductive reasoning and abductive reasoning). Below is an example of performing (deductive) reasoning, and users can refer to *Performing abductive reasoning in the knowledge base* for details of abductive reasoning.

```
pseudo_labels = [1, 2]
reasoning_result = kb.logic_forward(pseudo_labels)
print(f"Reasoning result of pseudo-labels {pseudo_labels} is {reasoning_result}.")
```

Out:

```
Reasoning result of pseudo-labels [1, 2] is 3.
```

Note: In addition to building a knowledge base based on `KBase`, we can also establish a knowledge base with a ground KB using `GroundKB`, or a knowledge base implemented based on Prolog files using `PrologKB`. The corresponding code for these implementations can be found in the `main.py` file. Those interested are encouraged to examine it for further insights.

Then, we create a reasoner by instantiating the class `Reasoner`. Due to the indeterminism of abductive reasoning, there could be multiple candidates compatible with the knowledge base. When this happens, reasoner can minimize inconsistencies between the knowledge base and pseudo-labels predicted by the learning part, and then return only one candidate that has the highest consistency.

```
reasoner = Reasoner(kb)
```

Note: During creating reasoner, the definition of “consistency” can be customized within the `dist_func` parameter. In the code above, we employ a consistency measurement based on confidence, which calculates the consistency between the data example and candidates based on the confidence derived from the predicted probability. In `examples/mnist_add/main.py`, we provide options for utilizing other forms of consistency measurement.

Also, during the process of inconsistency minimization, we can leverage [ZOOpt library](#) for acceleration. Options for this are also available in `examples/mnist_add/main.py`. Those interested are encouraged to explore these features.

2.10.4 Building Evaluation Metrics

Next, we set up evaluation metrics. These metrics will be used to evaluate the model performance during training and testing. Specifically, we use `SymbolAccuracy` and `ReasoningMetric`, which are used to evaluate the accuracy of the machine learning model's predictions and the accuracy of the final reasoning results, respectively.

```
metric_list = [SymbolAccuracy(prefix="mnist_add"), ReasoningMetric(kb=kb, prefix="mnist_
↪add")]
```

2.10.5 Bridging Learning and Reasoning

Now, the last step is to bridge the learning and reasoning part. We proceed with this step by creating an instance of `SimpleBridge`.

```
bridge = SimpleBridge(model, reasoner, metric_list)
```

Perform training and testing by invoking the `train` and `test` methods of `SimpleBridge`.

```
# Build logger
print_log("Abductive Learning on the MNIST Addition example.", logger="current")
log_dir = ABLLogger.get_current_instance().log_dir
weights_dir = osp.join(log_dir, "weights")

bridge.train(train_data, loops=1, segment_size=0.01, save_interval=1, save_dir=weights_
↪dir)
bridge.test(test_data)
```

The log will appear similar to the following:

Log:

```
abl - INFO - Abductive Learning on the MNIST Addition example.
abl - INFO - Working with Data.
abl - INFO - Building the Learning Part.
abl - INFO - Building the Reasoning Part.
abl - INFO - Building Evaluation Metrics.
abl - INFO - Bridge Learning and Reasoning.
abl - INFO - loop(train) [1/2] segment(train) [1/100]
abl - INFO - model loss: 2.25980
abl - INFO - loop(train) [1/2] segment(train) [2/100]
abl - INFO - model loss: 2.14168
abl - INFO - loop(train) [1/2] segment(train) [3/100]
abl - INFO - model loss: 2.02010
...
abl - INFO - loop(train) [2/2] segment(train) [1/100]
abl - INFO - model loss: 0.90260
...
abl - INFO - Eval start: loop(val) [2]
```

(continues on next page)

(continued from previous page)

```
abl - INFO - Evaluation ended, mnist_add/character_accuracy: 0.993 mnist_add/
↪reasoning_accuracy: 0.986
abl - INFO - Test start:
abl - INFO - Evaluation ended, mnist_add/character_accuracy: 0.991 mnist_add/
↪reasoning_accuracy: 0.980
```

2.10.6 Environment

For all experiments, we used a single linux server. Details on the specifications are listed in the table below.

2.10.7 Performance

We present the results of ABL as follows, which include the reasoning accuracy (the proportion of equations that are correctly summed), and the training time used to achieve this accuracy. These results are compared with the following methods:

- **NeurASP**: An extension of answer set programs by treating the neural network output as the probability distribution over atomic facts;
- **DeepProbLog**: An extension of ProbLog by introducing neural predicates in Probabilistic Logic Programming;
- **LTN**: A neural-symbolic framework that uses differentiable first-order logic language to incorporate data and logic;
- **DeepStochLog**: A neural-symbolic framework based on stochastic logic program.

2.11 Handwritten Formula (HWF)

Below shows an implementation of **Handwritten Formula**. In this task, handwritten images of decimal formulas and their computed results are given, alongwith a domain knowledge base containing information on how to compute the decimal formula. The task is to recognize the symbols (which can be digits or operators '+', '-', '×', '÷') of handwritten images and accurately determine their results.

Intuitively, we first use a machine learning model (learning part) to convert the input images to symbols (we call them pseudo-labels), and then use the knowledge base (reasoning part) to calculate the results of these symbols. Since we do not have ground-truth of the symbols, in Abductive Learning, the reasoning part will leverage domain knowledge and revise the initial symbols yielded by the learning part through abductive reasoning. This process enables us to further update the machine learning model.

```
# Import necessary libraries and modules
import os.path as osp

import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.nn as nn

from ablkit.bridge import SimpleBridge
from ablkit.data.evaluation import ReasoningMetric, SymbolAccuracy
from ablkit.learning import ABLModel, BasicNN
from ablkit.reasoning import KBBase, Reasoner
```

(continues on next page)

(continued from previous page)

```

from ablkit.utils import ABLLogger, print_log

from datasets import get_dataset
from models.nn import SymbolNet

```

2.11.1 Working with Data

First, we get the training and testing datasets:

```

train_data = get_dataset(train=True, get_pseudo_label=True)
test_data = get_dataset(train=False, get_pseudo_label=True)

```

Both `train_data` and `test_data` have the same structures: tuples with three components: `X` (list where each element is a list of images), `gt_pseudo_label` (list where each element is a list of symbols, i.e., pseudo-labels) and `Y` (list where each element is the computed result). The length and structures of datasets are illustrated as follows.

Note: `gt_pseudo_label` is only used to evaluate the performance of the learning part but not to train the model.

```

print(f"Both train_data and test_data consist of 3 components: X, gt_pseudo_label, Y")
print()
train_X, train_gt_pseudo_label, train_Y = train_data
print(f"Length of X, gt_pseudo_label, Y in train_data: " +
      f"{len(train_X)}, {len(train_gt_pseudo_label)}, {len(train_Y)}")
test_X, test_gt_pseudo_label, test_Y = test_data
print(f"Length of X, gt_pseudo_label, Y in test_data: " +
      f"{len(test_X)}, {len(test_gt_pseudo_label)}, {len(test_Y)}")
print()

X_0, gt_pseudo_label_0, Y_0 = train_X[0], train_gt_pseudo_label[0], train_Y[0]
print(f"X is a {type(train_X).__name__}, " +
      f"with each element being a {type(X_0).__name__} of {type(X_0[0]).__name__}.")
print(f"gt_pseudo_label is a {type(train_gt_pseudo_label).__name__}, " +
      f"with each element being a {type(gt_pseudo_label_0).__name__} " +
      f"of {type(gt_pseudo_label_0[0]).__name__}.")
print(f"Y is a {type(train_Y).__name__}, " +
      f"with each element being an {type(Y_0).__name__}.")

```

Out:

```

Both train_data and test_data consist of 3 components: X, gt_pseudo_label, Y

Length of X, gt_pseudo_label, Y in train_data: 10000, 10000, 10000
Length of X, gt_pseudo_label, Y in test_data: 2000, 2000, 2000

X is a list, with each element being a list of Tensor.
gt_pseudo_label is a list, with each element being a list of str.
Y is a list, with each element being an int.

```

The *i*th element of `X`, `gt_pseudo_label`, and `Y` together constitute the *i*th data example. Here we use two of them (the 1001st and the 3001st) as illustrations:

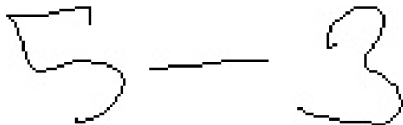
```

X_1000, gt_pseudo_label_1000, Y_1000 = train_X[1000], train_gt_pseudo_label[1000], train_
→Y[1000]
print(f"X in the 1001st data example (a list of images):")
for i, x in enumerate(X_1000):
    plt.subplot(1, len(X_1000), i+1)
    plt.axis('off')
    plt.imshow(x.squeeze(), cmap='gray')
plt.show()
print(f"gt_pseudo_label in the 1001st data example (a list of ground truth pseudo-
→labels): {gt_pseudo_label_1000}")
print(f"Y in the 1001st data example (the computed result): {Y_1000}")
print()
X_3000, gt_pseudo_label_3000, Y_3000 = train_X[3000], train_gt_pseudo_label[3000], train_
→Y[3000]
print(f"X in the 3001st data example (a list of images):")
for i, x in enumerate(X_3000):
    plt.subplot(1, len(X_3000), i+1)
    plt.axis('off')
    plt.imshow(x.squeeze(), cmap='gray')
plt.show()
print(f"gt_pseudo_label in the 3001st data example (a list of ground truth pseudo-
→labels): {gt_pseudo_label_3000}")
print(f"Y in the 3001st data example (the computed result): {Y_3000}")

```

Out:

X in the 1001st data example (a list of images):



gt_pseudo_label in the 1001st data example (a list of pseudo-labels): ['5', '-', '3'
→']

Y in the 1001st data example (the computed result): 2

X in the 3001st data example (a list of images):



gt_pseudo_label in the 3001st data example (a list of pseudo-labels): ['4', '/', '6'
→', '*', '5']

Y in the 3001st data example (the computed result): 3.3333333333333333

Note: The symbols in the HWF dataset can be one of digits or operators '+', '-', '×', '÷'.

We may see that, in the 1001st data example, the length of the formula is 3, while in the 3001st data example, the length of the formula is 5. In the HWF dataset, the lengths of the formulas are 1, 3, 5, and 7 (Specifically, 10% of the equations

have a length of 1, 10% have a length of 3, 20% have a length of 5, and 60% have a length of 7).

2.11.2 Building the Learning Part

To build the learning part, we need to first build a machine learning base model. We use SymbolNet, and encapsulate it within a BasicNN object to create the base model. BasicNN is a class that encapsulates a PyTorch model, transforming it into a base model with an sklearn-style interface.

```
# class of symbol may be one of ['1', ..., '9', '+', '-', '*', '/'], total of 14 classes
cls = SymbolNet(num_classes=13, image_size=(45, 45, 1))
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cls.parameters(), lr=0.001, betas=(0.9, 0.99))
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

base_model = BasicNN(
    model=cls,
    loss_fn=loss_fn,
    optimizer=optimizer,
    device=device,
    batch_size=128,
    num_epochs=3,
)
```

BasicNN offers methods like `predict` and `predict_proba`, which are used to predict the class index and the probabilities of each class for images. As shown below:

```
data_instances = [torch.randn(1, 45, 45) for _ in range(32)]
pred_idx = base_model.predict(X=data_instances)
print(f"Predicted class index for a batch of 32 instances: " +
      f"{type(pred_idx).__name__} with shape {pred_idx.shape}")
pred_prob = base_model.predict_proba(X=data_instances)
print(f"Predicted class probabilities for a batch of 32 instances: " +
      f"{type(pred_prob).__name__} with shape {pred_prob.shape}")
```

Out:

```
Predicted class index for a batch of 32 instances: ndarray with shape (32,)
Predicted class probabilities for a batch of 32 instances: ndarray with shape (32, 14)
```

However, the base model built above deals with instance-level data (i.e., individual images), and can not directly deal with example-level data (i.e., a list of images comprising the formula). Therefore, we wrap the base model into ABLModel, which enables the learning part to train, test, and predict on example-level data.

```
model = ABLModel(base_model)
```

As an illustration, consider this example of training on example-level data using the `predict` method in ABLModel. In this process, the method accepts data examples as input and outputs the class labels and the probabilities of each class for all instances within these data examples.

```
from ablkit.data.structures import ListData
# ListData is a data structure provided by ABLkit that can be used to organize data.
```

(continues on next page)

(continued from previous page)

```

→examples
data_examples = ListData()
# We use the first 1001st and 3001st data examples in the training set as an illustration
data_examples.X = [X_1000, X_3000]
data_examples.gt_pseudo_label = [gt_pseudo_label_1000, gt_pseudo_label_3000]
data_examples.Y = [Y_1000, Y_3000]

# Perform prediction on the two data examples
# Remind that, in the 1001st data example, the length of the formula is 3,
# while in the 3001st data example, the length of the formula is 5.
pred_label, pred_prob = model.predict(data_examples)['label'], model.predict(data_
→examples)['prob']
print(f"Predicted class labels for the 100 data examples: a list of length {len(pred_
→label)}, \n" +
      f"the first element is a {type(pred_label[0]).__name__} of shape {pred_label[0].
→shape}, " +
      f"and the second element is a {type(pred_label[1]).__name__} of shape {pred_
→label[1].shape}. \n")
print(f"Predicted class probabilities for the 100 data examples: a list of length
→{len(pred_prob)}, \n"
      f"the first element is a {type(pred_prob[0]).__name__} of shape {pred_prob[0].
→shape}, " +
      f"and the second element is a {type(pred_prob[1]).__name__} of shape {pred_prob[1].
→shape}.")

```

Out:

```

Predicted class labels for the 100 data examples: a list of length 2,
the first element is a ndarray of shape (3,), and the second element is a ndarray_
→of shape (5,).

Predicted class probabilities for the 100 data examples: a list of length 2,
the first element is a ndarray of shape (3, 14), and the second element is a_
→ndarray of shape (5, 14).

```

2.11.3 Building the Reasoning Part

In the reasoning part, we first build a knowledge base which contains information on how to compute a formula. We build it by creating a subclass of `KBBBase`. In the derived subclass, we initialize the `pseudo_label_list` parameter specifying list of possible pseudo-labels, and override the `logic_forward` function defining how to perform (deductive) reasoning.

```

class HwfKB(KBBBase):
    def __init__(self, pseudo_label_list=["1", "2", "3", "4", "5", "6", "7", "8", "9", "+
→", "-", "*", "/"]):
        super().__init__(pseudo_label_list)

    def _valid_candidate(self, formula):
        if len(formula) % 2 == 0:
            return False
        for i in range(len(formula)):

```

(continues on next page)

(continued from previous page)

```

        if i % 2 == 0 and formula[i] not in ["1", "2", "3", "4", "5", "6", "7", "8",
↪ "9"]:
            return False
        if i % 2 != 0 and formula[i] not in ["+", "-", "*", "/"]:
            return False
        return True

# Implement the deduction function
def logic_forward(self, formula):
    if not self._valid_candidate(formula):
        return np.inf
    return eval("".join(formula))

kb = HwfKB()

```

The knowledge base can perform logical reasoning (both deductive reasoning and abductive reasoning). Below is an example of performing (deductive) reasoning, and users can refer to *Performing abductive reasoning in the knowledge base* for details of abductive reasoning.

```

pseudo_labels = ["1", "-", "2", "*", "5"]
reasoning_result = kb.logic_forward(pseudo_labels)
print(f"Reasoning result of pseudo-labels {pseudo_labels} is {reasoning_result}.")

```

Out:

```
Reasoning result of pseudo-labels ['1', '-', '2', '*', '5'] is -9.
```

Note: In addition to building a knowledge base based on KBBase, we can also establish a knowledge base with a ground KB using GroundKB. The corresponding code can be found in the `examples/hwf/main.py` file. Those interested are encouraged to examine it for further insights.

Also, when building the knowledge base, we can also set the `max_err` parameter during initialization, which is shown in the `examples/hwf/main.py` file. This parameter specifies the upper tolerance limit when comparing the similarity between the reasoning result of pseudo-labels and the ground truth during abductive reasoning, with a default value of `1e-10`.

Then, we create a reasoner by instantiating the class `Reasoner`. Due to the indeterminism of abductive reasoning, there could be multiple candidates compatible with the knowledge base. When this happens, reasoner can minimize inconsistencies between the knowledge base and pseudo-labels predicted by the learning part, and then return only one candidate that has the highest consistency.

```
reasoner = Reasoner(kb)
```

Note: During creating reasoner, the definition of “consistency” can be customized within the `dist_func` parameter. In the code above, we employ a consistency measurement based on confidence, which calculates the consistency between the data example and candidates based on the confidence derived from the predicted probability. In `examples/hwf/main.py`, we provide options for utilizing other forms of consistency measurement.

Also, during the process of inconsistency minimization, we can leverage [ZOOpt library](#) for acceleration. Options for this are also available in `examples/hwf/main.py`. Those interested are encouraged to explore these features.

2.11.4 Building Evaluation Metrics

Next, we set up evaluation metrics. These metrics will be used to evaluate the model performance during training and testing. Specifically, we use `SymbolAccuracy` and `ReasoningMetric`, which are used to evaluate the accuracy of the machine learning model's predictions and the accuracy of the final reasoning results, respectively.

```
metric_list = [SymbolAccuracy(prefix="hwf"), ReasoningMetric(kb=kb, prefix="hwf")]
```

2.11.5 Bridging Learning and Reasoning

Now, the last step is to bridge the learning and reasoning part. We proceed with this step by creating an instance of `SimpleBridge`.

```
bridge = SimpleBridge(model, reasoner, metric_list)
```

Perform training and testing by invoking the `train` and `test` methods of `SimpleBridge`.

```
# Build logger
print_log("Abductive Learning on the HWF example.", logger="current")
log_dir = ABLLogger.get_current_instance().log_dir
weights_dir = osp.join(log_dir, "weights")

bridge.train(train_data, loops=3, segment_size=1000, save_dir=weights_dir)
bridge.test(test_data)
```

The log will appear similar to the following:

Log:

```
abl - INFO - Abductive Learning on the HWF example.
abl - INFO - loop(train) [1/3] segment(train) [1/10]
abl - INFO - model loss: 0.00024
abl - INFO - loop(train) [1/3] segment(train) [2/10]
abl - INFO - model loss: 0.00011
abl - INFO - loop(train) [1/3] segment(train) [3/10]
abl - INFO - model loss: 0.00332
...
abl - INFO - Eval start: loop(val) [1]
abl - INFO - Evaluation ended, hwf/character_accuracy: 0.997 hwf/reasoning_
↪accuracy: 0.985
abl - INFO - loop(train) [2/3] segment(train) [1/10]
abl - INFO - model loss: 0.00126
...
abl - INFO - Eval start: loop(val) [2]
abl - INFO - Evaluation ended, hwf/character_accuracy: 0.998 hwf/reasoning_
↪accuracy: 0.989
abl - INFO - loop(train) [3/3] segment(train) [1/10]
abl - INFO - model loss: 0.00030
...
abl - INFO - Eval start: loop(val) [3]
abl - INFO - Evaluation ended, hwf/character_accuracy: 0.999 hwf/reasoning_
↪accuracy: 0.996
abl - INFO - Test start:
```

(continues on next page)

(continued from previous page)

```
abl - INFO - Evaluation ended, hwf/character_accuracy: 0.997 hwf/reasoning_
↪accuracy: 0.986
```

2.11.6 Environment

For all experiments, we used a single linux server. Details on the specifications are listed in the table below.

2.11.7 Performance

We present the results of ABL as follows, which include the reasoning accuracy (for different equation lengths in the HWF dataset), training time (to achieve the accuracy using all equation lengths), and average memory usage (using all equation lengths). These results are compared with the following methods:

- **NGS**: A neural-symbolic framework that uses a grammar model and a back-search algorithm to improve its computing process;
- **DeepProbLog**: An extension of ProbLog by introducing neural predicates in Probabilistic Logic Programming;
- **DeepStochLog**: A neural-symbolic framework based on stochastic logic program.

2.12 Handwritten Equation Decipherment (HED)

Below shows an implementation of **Handwritten Equation Decipherment**. In this task, the handwritten equations are given, which consist of sequential pictures of characters. The equations are generated with unknown operation rules from images of symbols ('0', '1', '+' and '='), and each equation is associated with a label indicating whether the equation is correct (i.e., positive) or not (i.e., negative). Also, we are given a knowledge base which involves the structure of the equations and a recursive definition of bit-wise operations. The task is to learn from a training set of above-mentioned equations and then to predict labels of unseen equations.

Intuitively, we first use a machine learning model (learning part) to obtain the pseudo-labels ('0', '1', '+' and '=') for the observed pictures. We then use the knowledge base (reasoning part) to perform abductive reasoning so as to yield ground hypotheses as possible explanations to the observed facts, suggesting some pseudo-labels to be revised. This process enables us to further update the machine learning model.

```
# Import necessary libraries and modules
import os.path as osp

import matplotlib.pyplot as plt
import torch
import torch.nn as nn

from ablkit.learning import ABLModel, BasicNN
from ablkit.utils import ABLLogger, print_log

from bridge import HedBridge
from consistency_metric import ConsistencyMetric
from datasets import get_dataset, split_equation
from models.nn import SymbolNet
from reasoning import HedKB, HedReasoner
```


2.12.1 Working with Data

First, we get the datasets of handwritten equations:

```
total_train_data = get_dataset(train=True)
train_data, val_data = split_equation(total_train_data, 3, 1)
test_data = get_dataset(train=False)
```

The datasets are shown below:

```
true_train_equation = train_data[1]
false_train_equation = train_data[0]
print(f"Equations in the dataset is organized by equation length, " +
      f"from {min(train_data[0].keys())} to {max(train_data[0].keys()).}")
print()

true_train_equation_with_length_5 = true_train_equation[5]
false_train_equation_with_length_5 = false_train_equation[5]
print(f"For each equation length, there are {len(true_train_equation_with_length_5)} " +
      f"true equations and {len(false_train_equation_with_length_5)} false equations " +
      f"in the training set.")

true_val_equation = val_data[1]
false_val_equation = val_data[0]
true_val_equation_with_length_5 = true_val_equation[5]
false_val_equation_with_length_5 = false_val_equation[5]
print(f"For each equation length, there are {len(true_val_equation_with_length_5)} " +
      f"true equations and {len(false_val_equation_with_length_5)} false equations " +
      f"in the validation set.")

true_test_equation = test_data[1]
false_test_equation = test_data[0]
true_test_equation_with_length_5 = true_test_equation[5]
false_test_equation_with_length_5 = false_test_equation[5]
print(f"For each equation length, there are {len(true_test_equation_with_length_5)} " +
      f"true equations and {len(false_test_equation_with_length_5)} false equations " +
      f"in the test set.")
```

Out:

```
Equations in the dataset is organized by equation length, from 5 to 26.

For each equation length, there are 225 true equations and 225 false equations in_
↳ the training set.
For each equation length, there are 75 true equations and 75 false equations in the_
↳ validation set.
For each equation length, there are 300 true equations and 300 false equations in_
↳ the test set.
```

As illustrations, we show four equations in the training dataset:

```
true_train_equation_with_length_5 = true_train_equation[5]
true_train_equation_with_length_8 = true_train_equation[8]
print(f"First true equation with length 5 in the training dataset:")
```

(continues on next page)

(continued from previous page)

```

for i, x in enumerate(true_train_equation_with_length_5[0]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(x.squeeze(), cmap='gray')
plt.show()
print(f"First true equation with length 8 in the training dataset:")
for i, x in enumerate(true_train_equation_with_length_8[0]):
    plt.subplot(1, 8, i+1)
    plt.axis('off')
    plt.imshow(x.squeeze(), cmap='gray')
plt.show()

false_train_equation_with_length_5 = false_train_equation[5]
false_train_equation_with_length_8 = false_train_equation[8]
print(f"First false equation with length 5 in the training dataset:")
for i, x in enumerate(false_train_equation_with_length_5[0]):
    plt.subplot(1, 5, i+1)
    plt.axis('off')
    plt.imshow(x.squeeze(), cmap='gray')
plt.show()
print(f"First false equation with length 8 in the training dataset:")
for i, x in enumerate(false_train_equation_with_length_8[0]):
    plt.subplot(1, 8, i+1)
    plt.axis('off')
    plt.imshow(x.squeeze(), cmap='gray')
plt.show()

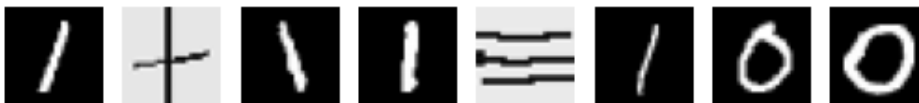
```

Out:

First true equation with length 5 in the training dataset:



First true equation with length 8 in the training dataset:



First false equation with length 5 in the training dataset:



First false equation with length 8 in the training dataset:



2.12.2 Building the Learning Part

To build the learning part, we need to first build a machine learning base model. We use SymbolNet, and encapsulate it within a BasicNN object to create the base model. BasicNN is a class that encapsulates a PyTorch model, transforming it into a base model with an sklearn-style interface.

```
# class of symbol may be one of ['0', '1', '+', '='], total of 4 classes
cls = SymbolNet(num_classes=4)
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.RMSprop(cls.parameters(), lr=0.001, weight_decay=1e-4)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

base_model = BasicNN(
    cls,
    loss_fn,
    optimizer,
    device=device,
    batch_size=32,
    num_epochs=1,
    stop_loss=None,
)
```

However, the base model built above deals with instance-level data (i.e., individual images), and can not directly deal with example-level data (i.e., a list of images comprising the equation). Therefore, we wrap the base model into ABLModel, which enables the learning part to train, test, and predict on example-level data.

```
model = ABLModel(base_model)
```

2.12.3 Building the Reasoning Part

In the reasoning part, we first build a knowledge base. As mentioned before, the knowledge base in this task involves the structure of the equations and a recursive definition of bit-wise operations, which are defined in Prolog file `examples/hed/reasoning/BK.pl` and `examples/hed/reasoning/learn_add.pl`, respectively. Specifically, the knowledge about the structure of equations is a set of DCG rules recursively define that a digit is a sequence of '0' and '1', and equations share the structure of $X+Y=Z$, though the length of X , Y and Z can be varied. The knowledge about bit-wise operations is a recursive logic program, which reversely calculates $X+Y$, i.e., it operates on X and Y digit-by-digit and from the last digit to the first.

The knowledge base is already built in HedKB. HedKB is derived from class PrologKB, and is built upon the aforementioned Prolog files.

```
kb = HedKB()
```

Note: Please notice that, the specific rules for calculating the operations are undefined in the knowledge base, i.e., results of '0+0', '0+1' and '1+1' could be '0', '1', '00', '01' or even '10'. The missing calculation rules are required to

be learned from the data. Therefore, HedKB incorporates methods for abducting rules from data. Users interested can refer to the specific implementation of HedKB in `examples/hed/reasoning/reasoning.py`

Then, we create a reasoner. Due to the indeterminism of abductive reasoning, there could be multiple candidates compatible with the knowledge base. When this happens, reasoner can minimize inconsistencies between the knowledge base and pseudo-labels predicted by the learning part, and then return only one candidate that has the highest consistency.

In this task, we create the reasoner by instantiating the class `HedReasoner`, which is a reasoner derived from `Reasoner` and tailored specifically for this task. `HedReasoner` leverages `ZOOpt library` for acceleration, and has designed a specific strategy to better harness `ZOOpt`'s capabilities. Additionally, methods for abducting rules from data have been incorporated. Users interested can refer to the specific implementation of `HedReasoner` in `reasoning/reasoning.py`.

```
reasoner = HedReasoner(kb, dist_func="hamming", use_zoopt=True, max_revision=10)
```

2.12.4 Building Evaluation Metrics

Next, we set up evaluation metrics. These metrics will be used to evaluate the model performance during training and testing. Specifically, we use `SymbolAccuracy` and `ReasoningMetric`, which are used to evaluate the accuracy of the machine learning model's predictions and the accuracy of the final reasoning results, respectively.

```
# Set up metrics
metric_list = [SymbolAccuracy(prefix="hed"), ReasoningMetric(kb=kb, prefix="hed")]
```

2.12.5 Bridging Learning and Reasoning

Now, the last step is to bridge the learning and reasoning part. We proceed with this step by creating an instance of `HedBridge`, which is derived from `SimpleBridge` and tailored specific for this task.

```
bridge = HedBridge(model, reasoner, metric_list)
```

Perform pretraining, training and testing by invoking the `pretrain`, `train` and `test` methods of `HedBridge`.

```
# Build logger
print_log("Abductive Learning on the HED example.", logger="current")

# Retrieve the directory of the Log file and define the directory for saving the model_
↳ weights.
log_dir = ABLLogger.get_current_instance().log_dir
weights_dir = osp.join(log_dir, "weights")

bridge.pretrain("./weights")
bridge.train(train_data, val_data, save_dir=weights_dir)
bridge.test(test_data)
```

2.13 Zoo

Below shows an implementation of `Zoo` dataset. In this task, attributes of animals (such as presence of hair, eggs, etc.) and their targets (the animal class they belong to) are given, along with a knowledge base which contains information about the relations between attributes and targets, e.g., `Implies(milk == 1, mammal == 1)`.

The goal of this task is to develop a learning model that can predict the targets of animals based on their attributes. In the initial stages, when the model is under-trained, it may produce incorrect predictions that conflict with the relations contained in the knowledge base. When this happens, abductive reasoning can be employed to adjust these results and retrain the model accordingly. This process enables us to further update the learning model.

```
# Import necessary libraries and modules
import os.path as osp

import numpy as np
from sklearn.ensemble import RandomForestClassifier

from ablkit.bridge import SimpleBridge
from ablkit.data.evaluation import ReasoningMetric, SymbolAccuracy
from ablkit.learning import ABLModel
from ablkit.reasoning import Reasoner
from ablkit.utils import ABLLogger, confidence_dist, print_log, tab_data_to_tuple

from get_dataset import load_and_preprocess_dataset, split_dataset
from kb import ZooKB
```

2.13.1 Working with Data

First, we load and preprocess the `Zoo` dataset, and split it into labeled/unlabeled/test data

```
X, y = load_and_preprocess_dataset(dataset_id=62)
X_label, y_label, X_unlabel, y_unlabel, X_test, y_test = split_dataset(X, y, test_size=0.
↪ 3)
```

`Zoo` dataset consists of tabular data. The attributes contain 17 boolean values (e.g., hair, feathers, eggs, milk, airborne, aquatic, etc.) and the target is an integer value in the range [0,6] representing 7 classes (e.g., mammal, bird, reptile, fish, amphibian, insect, and other). Below is an illustration:

```
print("Shape of X and y:", X.shape, y.shape)
print("First five elements of X:")
print(X[:5])
print("First five elements of y:")
print(y[:5])
```

Out:

```
Shape of X and y: (101, 16) (101,)
First five elements of X:
[[True False False True False False True True True True False False 4
  False False True]
 [True False False True False False False True True True False False 4
  True False True]
```

(continues on next page)

(continued from previous page)

```
[False False True False False True True True True False False True 0
True False False]
[True False False True False False True True True True False False 4
False False True]
[True False False True False False True True True True False False 4
True False True]]
First five elements of y:
[0 0 3 0 0]
```

Next, we transform the tabular data to the format required by ABLkit, which is a tuple of (X, gt_pseudo_label, Y). In this task, we treat the attributes as X and the targets as gt_pseudo_label (ground truth pseudo-labels). Y (reasoning results) are expected to be 0, indicating no rules are violated.

```
label_data = tab_data_to_tuple(X_label, y_label, reasoning_result = 0)
data = tab_data_to_tuple(X_test, y_test, reasoning_result = 0)
train_data = tab_data_to_tuple(X_unlabel, y_unlabel, reasoning_result = 0)
```

2.13.2 Building the Learning Part

To build the learning part, we need to first build a machine learning base model. We use a [Random Forest](#) as the base model.

```
base_model = RandomForestClassifier()
```

However, the base model built above deals with instance-level data, and can not directly deal with example-level data. Therefore, we wrap the base model into ABLModel, which enables the learning part to train, test, and predict on example-level data.

```
model = ABLModel(base_model)
```

2.13.3 Building the Reasoning Part

In the reasoning part, we first build a knowledge base which contains information about the relations between attributes (X) and targets (pseudo-labels), e.g., Implies(milk == 1, mammal == 1). The knowledge base is built in the ZooKB class within file `examples/zoo/kb.py`, and is derived from the KBBBase class.

```
kb = ZooKB()
```

As mentioned, for all attributes and targets in the dataset, the reasoning results are expected to be 0 since there should be no violations of the established knowledge in real data. As shown below:

```
for idx, (x, y_item) in enumerate(zip(X[:5], y[:5])):
    print(f"Example {idx}: the attributes are: {x}, and the target is {y_item}.")
    print(f"Reasoning result is {kb.logic_forward([y_item], [x])}.")
    print()
```

Out:

```
Example 0: the attributes are: [True False False True False False True True True,
↪ True False False 4 False
```

(continues on next page)

(continued from previous page)

False True], and the target is 0.
Reasoning result is 0.

Example 1: the attributes are: [True False False True False False False True True,
↪ True False False 4 True
False True], and the target is 0.
Reasoning result is 0.

Example 2: the attributes are: [False False True False False True True True True,
↪ False False True 0 True
False False], and the target is 3.
Reasoning result is 0.

Example 3: the attributes are: [True False False True False False True True True,
↪ True False False 4 False
False True], and the target is 0.
Reasoning result is 0.

Example 4: the attributes are: [True False False True False False True True True,
↪ True False False 4 True
False True], and the target is 0.
Reasoning result is 0.

Then, we create a reasoner by instantiating the class `Reasoner`. Due to the indeterminism of abductive reasoning, there could be multiple candidates compatible with the knowledge base. When this happens, reasoner can minimize inconsistencies between the knowledge base and pseudo-labels predicted by the learning part, and then return only one candidate that has the highest consistency.

```
def consistency(data_example, candidates, candidate_idx, reasoning_results):
    pred_prob = data_example.pred_prob
    model_scores = confidence_dist(pred_prob, candidate_idx)
    rule_scores = np.array(reasoning_results)
    scores = model_scores + rule_scores
    return scores

reasoner = Reasoner(kb, dist_func=consistency)
```

2.13.4 Building Evaluation Metrics

Next, we set up evaluation metrics. These metrics will be used to evaluate the model performance during training and testing. Specifically, we use `SymbolAccuracy` and `ReasoningMetric`, which are used to evaluate the accuracy of the machine learning model's predictions and the accuracy of the final reasoning results, respectively.

```
metric_list = [SymbolAccuracy(prefix="zoo"), ReasoningMetric(kb=kb, prefix="zoo")]
```

2.13.5 Bridging Learning and Reasoning

Now, the last step is to bridge the learning and reasoning part. We proceed with this step by creating an instance of `SimpleBridge`.

```
bridge = SimpleBridge(model, reasoner, metric_list)
```

Perform training and testing by invoking the `train` and `test` methods of `SimpleBridge`.

```
# Build logger
print_log("Abductive Learning on the Zoo example.", logger="current")
log_dir = ABLLogger.get_current_instance().log_dir
weights_dir = osp.join(log_dir, "weights")

print_log("----- Use labeled data to pretrain the model -----", logger="current")
base_model.fit(X_label, y_label)
print_log("----- Test the initial model -----", logger="current")
bridge.test(test_data)
print_log("----- Use ABL to train the model -----", logger="current")
bridge.train(train_data=train_data, label_data=label_data, loops=3, segment_size=len(X_
↪unlabel), save_dir=weights_dir)
print_log("----- Test the final model -----", logger="current")
bridge.test(test_data)
```

The log will appear similar to the following:

Log:

```
abl - INFO - Abductive Learning on the ZOO example.
abl - INFO - ----- Use labeled data to pretrain the model -----
abl - INFO - ----- Test the initial model -----
abl - INFO - Evaluation ended, zoo/character_accuracy: 0.903 zoo/reasoning_
↪accuracy: 0.903
abl - INFO - ----- Use ABL to train the model -----
abl - INFO - loop(train) [1/3] segment(train) [1/1]
abl - INFO - Evaluation start: loop(val) [1]
abl - INFO - Evaluation ended, zoo/character_accuracy: 1.000 zoo/reasoning_
↪accuracy: 1.000
abl - INFO - loop(train) [2/3] segment(train) [1/1]
abl - INFO - Evaluation start: loop(val) [2]
abl - INFO - Evaluation ended, zoo/character_accuracy: 1.000 zoo/reasoning_
↪accuracy: 1.000
abl - INFO - loop(train) [3/3] segment(train) [1/1]
abl - INFO - Evaluation start: loop(val) [3]
abl - INFO - Evaluation ended, zoo/character_accuracy: 1.000 zoo/reasoning_
↪accuracy: 1.000
abl - INFO - ----- Test the final model -----
abl - INFO - Evaluation ended, zoo/character_accuracy: 0.968 zoo/reasoning_
↪accuracy: 0.968
```

We may see from the results, after undergoing training with ABL, the model's accuracy has improved.

2.14 ablkit.data

2.14.1 structures

class ablkit.data.structures.ListData(*, metainfo: Optional[dict] = None, **kwargs)

Bases: BaseDataElement

Abstract Data Interface used throughout the ABLkit.

ListData is the underlying data structure used in the ABLkit, designed to manage diverse forms of data dynamically generated throughout the Abductive Learning (ABL) framework. This includes handling raw data, predicted pseudo-labels, abduced pseudo-labels, pseudo-label indices, etc.

As a fundamental data structure in ABL, ListData is essential for the smooth transfer and manipulation of data across various components of the ABL framework, such as prediction, abductive reasoning, and training phases. It provides a unified data format across these stages, ensuring compatibility and flexibility in handling diverse data forms in the ABL framework.

The attributes in ListData are divided into two parts, the metainfo and the data respectively.

- **metainfo**: Usually used to store basic information about data examples, such as symbol number, image size, etc. The attributes can be accessed or modified by dict-like or object-like operations, such as `.` (for data access and modification), `in`, `del`, `pop(str)`, `get(str)`, `metainfo_keys()`, `metainfo_values()`, `metainfo_items()`, `set_metainfo()` (for set or change key-value pairs in metainfo).
- **data**: raw data, labels, predictions, and abduced results are stored. The attributes can be accessed or modified by dict-like or object-like operations, such as `.`, `in`, `del`, `pop(str)`, `get(str)`, `keys()`, `values()`, `items()`. Users can also apply tensor-like methods to all `torch.Tensor` in the `data_fields`, such as `.cuda()`, `.cpu()`, `.numpy()`, `.to()`, `to_tensor()`, `.detach()`.

ListData supports index and slice for data field. The type of value in data field can be either `None` or list of base data structures such as `torch.Tensor`, `numpy.ndarray`, `list`, `str` and `tuple`.

This design is inspired by and extends the functionalities of the `BaseDataElement` class implemented in `MMEngine`.

Examples

```
>>> from ablkit.data.structures import ListData
>>> import numpy as np
>>> import torch
>>> data_examples = ListData()
>>> data_examples.X = [list(torch.randn(2)) for _ in range(3)]
>>> data_examples.Y = [1, 2, 3]
>>> data_examples.gt_pseudo_label = [[1, 2], [3, 4], [5, 6]]
>>> len(data_examples)
3
>>> print(data_examples)
<ListData(
  META INFORMATION
  DATA FIELDS
  Y: [1, 2, 3]
  gt_pseudo_label: [[1, 2], [3, 4], [5, 6]]
  X: [[tensor(1.1949), tensor(-0.9378)], [tensor(0.7414), tensor(0.7603)],
  ↪ [tensor(1.0587), tensor(1.9697)]]
```

(continues on next page)

(continued from previous page)

```

) at 0x7f3bbf1991c0>
>>> print(data_examples[:1])
<ListData(
  META INFORMATION
  DATA FIELDS
  Y: [1]
  gt_pseudo_label: [[1, 2]]
  X: [[tensor(1.1949), tensor(-0.9378)]]
) at 0x7f3bbf1a3580>
>>> print(data_examples.elements_num("X"))
6
>>> print(data_examples.flatten("gt_pseudo_label"))
[1, 2, 3, 4, 5, 6]
>>> print(data_examples.to_tuple("Y"))
(1, 2, 3)

```

elements_num(*item: str*) → int

Return the number of elements in the attribute specified by *item*.

Parameters

item (*str*) – Name of the attribute for which the number of elements is to be determined.

Returns

The number of elements in the attribute specified by *item*.

Return type

int

flatten(*item: str*) → List

Flatten the list of the attribute specified by *item*.

Parameters

item – Name of the attribute to be flattened.

Returns

The flattened list of the attribute specified by *item*.

Return type

list

to_tuple(*item: str*) → tuple

Convert the attribute specified by *item* to a tuple.

Parameters

item (*str*) – Name of the attribute to be converted.

Returns

The attribute after conversion to a tuple.

Return type

tuple

2.14.2 evaluation

class `ablkit.data.evaluation.BaseMetric`(*prefix*: *Optional[str]* = *None*)

Bases: `object`

Base class for a metrics.

The metrics first processes each batch of `data_examples` and appends the processed results to the results list. Then, it computes the metrics of the entire dataset.

Parameters

prefix (*str*, *optional*) – The prefix that will be added in the metrics names to disambiguate homonymous metrics of different tasks. If `prefix` is not provided in the argument, `self.default_prefix` will be used instead. Defaults to `None`.

abstract `compute_metrics()` → `dict`

Compute the metrics from processed results.

Returns

The computed metrics. The keys are the names of the metrics, and the values are the corresponding results.

Return type

`dict`

evaluate() → `dict`

Evaluate the model performance of the whole dataset after processing all batches.

Returns

Evaluation metrics dict on the val dataset. The keys are the names of the metrics, and the values are the corresponding results.

Return type

`dict`

abstract `process`(*data_examples*: `ListData`) → `None`

Process one batch of data examples. The processed results should be stored in `self.results`, which will be used to compute the metrics when all batches have been processed.

Parameters

data_examples (`ListData`) – A batch of data examples.

class `ablkit.data.evaluation.ReasoningMetric`(*kb*: `KBBase`, *prefix*: *Optional[str]* = *None*)

Bases: `BaseMetric`

A metrics class for evaluating the model performance on tasks that need reasoning.

This class is designed to calculate the accuracy of the reasoning results. Reasoning results are generated by first using the learning part to predict pseudo-labels and then using a knowledge base (KB) to perform logical reasoning. The reasoning results are then compared with the ground truth to calculate the accuracy.

Parameters

- **kb** (`KBBase`) – An instance of a knowledge base, used for logical reasoning and validation. If not provided, reasoning checks are not performed. Defaults to `None`.
- **prefix** (*str*, *optional*) – The prefix that will be added to the metrics names to disambiguate homonymous metrics of different tasks. Inherits from `BaseMetric`. Defaults to `None`.

Notes

The *ReasoningMetric* expects `data_examples` to have the attributes *pred_pseudo_label*, *Y*, and *X*, corresponding to the predicted pseudo labels, ground truth of reasoning results, and input data, respectively.

compute_metrics() → dict

Compute the reasoning accuracy metrics from `self.results`. It calculates the percentage of correctly reasoned examples over all examples.

Returns

A dictionary containing the computed metrics. It includes the key 'reasoning_accuracy' which maps to the calculated reasoning accuracy, represented as a float between 0 and 1.

Return type

dict

process(*data_examples*: ListData) → None

Process a batch of data examples.

This method takes in a batch of data examples, each containing predicted pseudo-labels (*pred_pseudo_label*), ground truth of reasoning results (*Y*), and input data (*X*). It evaluates the reasoning accuracy of each example by comparing the logical reasoning result (derived using the knowledge base) of the predicted pseudo-labels against *Y*. The result of this comparison (1 for correct reasoning, 0 for incorrect) is appended to `self.results`.

Parameters

data_examples (ListData) – A batch of data examples.

class `ablkit.data.evaluation.SymbolAccuracy`(*prefix*: Optional[str] = None)

Bases: *BaseMetric*

A metrics class for evaluating symbol-level accuracy.

This class is designed to assess the accuracy of symbol prediction. Symbol accuracy is calculated by comparing predicted pseudo labels and their ground truth.

Parameters

prefix (*str*, *optional*) – The prefix that will be added to the metrics names to disambiguate homonymous metrics of different tasks. Inherits from *BaseMetric*. Defaults to None.

compute_metrics() → dict

Compute the symbol accuracy metrics from `self.results`. It calculates the percentage of correctly predicted pseudo-labels over all pseudo-labels.

Returns

A dictionary containing the computed metrics. It includes the key 'character_accuracy' which maps to the calculated symbol-level accuracy, represented as a float between 0 and 1.

Return type

dict

process(*data_examples*: ListData) → None

Processes a batch of data examples.

This method takes in a batch of data examples, each containing a list of predicted pseudo-labels (*pred_pseudo_label*) and their ground truth (*gt_pseudo_label*). It calculates the accuracy by comparing the two lists. Then, a tuple of correct symbol count and total symbol count is appended to `self.results`.

Parameters

data_examples (ListData) – A batch of data examples, each containing: -

`pred_pseudo_label`: List of predicted pseudo-labels. - `gt_pseudo_label`: List of ground truth pseudo-labels.

Raises

ValueError – If the lengths of predicted and ground truth symbol lists are not equal.

2.15 ablkit.learning

class `ablkit.learning.ABLModel`(*base_model: Any*)

Bases: `object`

Serialize data and provide a unified interface for different machine learning models.

Parameters

base_model (*Machine Learning Model*) – The machine learning base model used for training and prediction. This model should implement the `fit` and `predict` methods. It's recommended, but not required, for the model to also implement the `predict_proba` method for generating predictions on the probabilities.

load(*args, **kwargs) → `None`

Load the model from a file.

This method delegates to the `load` method of `self.base_model`. The arguments passed to this method should match those expected by the `load` method of `self.base_model`.

predict(*data_examples: ListData*) → `Dict`

Predict the labels and probabilities for the given data.

Parameters

data_examples (*ListData*) – A batch of data to predict on.

Returns

A dictionary containing the predicted labels and probabilities.

Return type

`dict`

save(*args, **kwargs) → `None`

Save the model to a file.

This method delegates to the `save` method of `self.base_model`. The arguments passed to this method should match those expected by the `save` method of `self.base_model`.

train(*data_examples: ListData*) → `float`

Train the model on the given data.

Parameters

data_examples (*ListData*) – A batch of data to train on, which typically contains the data, `X`, and the corresponding labels, `abduced_idx`.

Returns

The loss value of the trained model.

Return type

`float`

valid(*data_examples: ListData*) → `float`

Validate the model on the given data.

Parameters

data_examples (*ListData*) – A batch of data to train on, which typically contains the data, X, and the corresponding labels, *abduced_idx*.

Returns

The accuracy of the trained model.

Return type

float

```
class ablkit.learning.BasicNN(model: Module, loss_fn: Module, optimizer: Optimizer, scheduler:  
    Optional[Callable[[...], Any]] = None, device: Union[device, str] =  
    device(type='cpu'), batch_size: int = 32, num_epochs: int = 1, stop_loss:  
    Optional[float] = 0.0001, num_workers: int = 0, save_interval: Optional[int]  
    = None, save_dir: Optional[str] = None, train_transform:  
    Optional[Callable[[...], Any]] = None, test_transform:  
    Optional[Callable[[...], Any]] = None, collate_fn:  
    Optional[Callable[[List[Any]], Any]] = None)
```

Bases: object

Wrap NN models into the form of an sklearn estimator.

Parameters

- **model** (*torch.nn.Module*) – The PyTorch model to be trained or used for prediction.
- **loss_fn** (*torch.nn.Module*) – The loss function used for training.
- **optimizer** (*torch.optim.Optimizer*) – The optimizer used for training.
- **scheduler** (*Callable[..., Any], optional*) – The learning rate scheduler used for training, which will be called at the end of each run of the `fit` method. It should implement the `step` method. Defaults to None.
- **device** (*Union[torch.device, str]*) – The device on which the model will be trained or used for prediction, Defaults to `torch.device("cpu")`.
- **batch_size** (*int, optional*) – The batch size used for training. Defaults to 32.
- **num_epochs** (*int, optional*) – The number of epochs used for training. Defaults to 1.
- **stop_loss** (*float, optional*) – The loss value at which to stop training. Defaults to 0.0001.
- **num_workers** (*int*) – The number of workers used for loading data. Defaults to 0.
- **save_interval** (*int, optional*) – The model will be saved every `save_interval` epoch during training. Defaults to None.
- **save_dir** (*str, optional*) – The directory in which to save the model during training. Defaults to None.
- **train_transform** (*Callable[..., Any], optional*) – A function/transform that takes an object and returns a transformed version used in the `fit` and `train_epoch` methods. Defaults to None.
- **test_transform** (*Callable[..., Any], optional*) – A function/transform that takes an object and returns a transformed version in the `predict`, `predict_proba` and `score` methods. Defaults to None.
- **collate_fn** (*Callable[[List[T]], Any], optional*) – The function used to collate data. Defaults to None.

fit(*data_loader*: *Optional[DataLoader] = None*, *X*: *Optional[List[Any]] = None*, *y*: *Optional[List[int]] = None*) → *BasicNN*

Train the model for `self.num_epochs` times or until the average loss on one epoch is less than `self.stop_loss`. It supports training with either a `DataLoader` object (`data_loader`) or a pair of input data (`X`) and target labels (`y`). If both `data_loader` and (`X`, `y`) are provided, the method will prioritize using the `data_loader`.

Parameters

- **data_loader** (*DataLoader*, *optional*) – The data loader used for training. Defaults to `None`.
- **X** (*List[Any]*, *optional*) – The input data. Defaults to `None`.
- **y** (*List[int]*, *optional*) – The target data. Defaults to `None`.

Returns

The model itself after training.

Return type

BasicNN

load(*load_path*: *str*) → `None`

Load the model and the optimizer.

Parameters

load_path (*str*) – The directory to load the model. Defaults to `""`.

predict(*data_loader*: *Optional[DataLoader] = None*, *X*: *Optional[List[Any]] = None*) → `ndarray`

Predict the class of the input data. This method supports prediction with either a `DataLoader` object (`data_loader`) or a list of input data (`X`). If both `data_loader` and `X` are provided, the method will predict the input data in `data_loader` instead of `X`.

Parameters

- **data_loader** (*DataLoader*, *optional*) – The data loader used for prediction. Defaults to `None`.
- **X** (*List[Any]*, *optional*) – The input data. Defaults to `None`.

Returns

The predicted class of the input data.

Return type

`numpy.ndarray`

predict_proba(*data_loader*: *Optional[DataLoader] = None*, *X*: *Optional[List[Any]] = None*) → `ndarray`

Predict the probability of each class for the input data. This method supports prediction with either a `DataLoader` object (`data_loader`) or a list of input data (`X`). If both `data_loader` and `X` are provided, the method will predict the input data in `data_loader` instead of `X`.

Parameters

- **data_loader** (*DataLoader*, *optional*) – The data loader used for prediction. Defaults to `None`.
- **X** (*List[Any]*, *optional*) – The input data. Defaults to `None`.

Warning: This method calculates the probability by applying a softmax function to the output of the neural network. If your neural network already includes a softmax function as its final activation, applying softmax again here will lead to incorrect probabilities.

Returns

The predicted probability of each class for the input data.

Return type

numpy.ndarray

save(*epoch_id*: int = 0, *save_path*: Optional[str] = None) → None

Save the model and the optimizer. User can either provide a *save_path* or specify the *epoch_id* at which the model and optimizer is saved. If both *save_path* and *epoch_id* are provided, *save_path* will be used. If only *epoch_id* is specified, model and optimizer will be saved to the path `f"model_checkpoint_epoch_{epoch_id}.pth"` under `self.save_dir`. *save_path* and *epoch_id* can not be None simultaneously.

Parameters

- **epoch_id** (int) – The epoch id.
- **save_path** (str, optional) – The path to save the model. Defaults to None.

score(*data_loader*: Optional[DataLoader] = None, *X*: Optional[List[Any]] = None, *y*: Optional[List[int]] = None) → float

Validate the model. It supports validation with either a `DataLoader` object (*data_loader*) or a pair of input data (*X*) and ground truth labels (*y*). If both *data_loader* and (*X*, *y*) are provided, the method will prioritize using the *data_loader*.

Parameters

- **data_loader** (DataLoader, optional) – The data loader used for scoring. Defaults to None.
- **X** (List[Any], optional) – The input data. Defaults to None.
- **y** (List[int], optional) – The target data. Defaults to None.

Returns

The accuracy of the model.

Return type

float

train_epoch(*data_loader*: DataLoader) → float

Train the model with an instance of `DataLoader` (*data_loader*) for one epoch.

Parameters

- **data_loader** (DataLoader) – The data loader used for training.

Returns

The average loss on one epoch.

Return type

float

2.15.1 torch_dataset

```
class ablkit.learning.torch_dataset.ClassificationDataset(X: List[Any], Y: List[int], transform:
Optional[Callable[[...], Any]] = None)
```

Bases: Dataset

Dataset used for classification task.

Parameters

- **X** (*List*[Any]) – The input data.
- **Y** (*List*[int]) – The target data.
- **transform** (*Callable*[..., Any], *optional*) – A function/transform that takes an object and returns a transformed version. Defaults to None.

```
class ablkit.learning.torch_dataset.PredictionDataset(X: List[Any], transform:
Optional[Callable[[...], Any]] = None)
```

Bases: Dataset

Dataset used for prediction.

Parameters

- **X** (*List*[Any]) – The input data.
- **transform** (*Callable*[..., Any], *optional*) – A function/transform that takes an object and returns a transformed version. Defaults to None.

```
class ablkit.learning.torch_dataset.RegressionDataset(X: List[Any], Y: List[Any])
```

Bases: Dataset

Dataset used for regression task.

Parameters

- **X** (*List*[Any]) – A list of objects representing the input data.
- **Y** (*List*[Any]) – A list of objects representing the output data.

2.16 ablkit.reasoning

```
class ablkit.reasoning.GroundKB(pseudo_label_list: List[Any], GKB_len_list: List[int], max_err: float =
1e-10)
```

Bases: [KBBase](#)

Knowledge base with a ground KB (GKB). Ground KB is a knowledge base prebuilt upon class initialization, storing all potential candidates along with their respective reasoning result. Ground KB can accelerate abductive reasoning in `abduce_candidates`.

Parameters

- **pseudo_label_list** (*List*[Any]) – Refer to class [KBBase](#).
- **GKB_len_list** (*List*[int]) – List of possible lengths for pseudo-labels of an example.
- **max_err** (*float*, *optional*) – Refer to class [KBBase](#).

Notes

Users can also inherit from this class to build their own knowledge base. Similar to `KBBase`, users are only required to provide the `pseudo_label_list` and override the `logic_forward` function. Additionally, users should provide the `GKB_len_list`. After that, other operations (e.g. auto-construction of GKB, and how to perform abductive reasoning) will be automatically set up.

abduce_candidates(*pseudo_label*: *List[Any]*, *y*: *Any*, *x*: *List[Any]*, *max_revision_num*: *int*, *require_more_revision*: *int*) → *List[List[Any]]*

Perform abductive reasoning by directly retrieving compatible candidates from the prebuilt GKB. In this way, the time-consuming exhaustive search can be avoided.

Parameters

- **pseudo_label** (*List[Any]*) – Pseudo-labels of an example (to be revised by abductive reasoning).
- **y** (*Any*) – Ground truth of the reasoning result for the example.
- **x** (*List[Any]*) – The example (unused in `GroundKB`).
- **max_revision_num** (*int*) – The upper limit on the number of revised labels for each example.
- **require_more_revision** (*int*) – Specifies additional number of revisions permitted beyond the minimum required.

Returns

A tuple of two elements. The first element is a list of candidate revisions, i.e. revised pseudo-labels of the example. that are compatible with the knowledge base. The second element is a list of reasoning results corresponding to each candidate, i.e., the outcome of the `logic_forward` function.

Return type

Tuple[List[List[Any]], List[Any]]

```
class ablkit.reasoning.KBBase(pseudo_label_list: ~typing.List[~typing.Any], max_err: float = 1e-10,
                             use_cache: bool = True, key_func: ~typing.Callable = <function
                             to_hashable>, cache_size: int = 4096)
```

Bases: `ABC`

Base class for knowledge base.

Parameters

- **pseudo_label_list** (*List[Any]*) – List of possible pseudo-labels. It's recommended to arrange the pseudo-labels in this list so that each aligns with its corresponding index in the base model: the first with the 0th index, the second with the 1st, and so forth.
- **max_err** (*float*, *optional*) – The upper tolerance limit when comparing the similarity between the reasoning result of pseudo-labels and the ground truth. This is only applicable when the reasoning result is of a numerical type. This is particularly relevant for regression problems where exact matches might not be feasible. Defaults to 1e-10.
- **use_cache** (*bool*, *optional*) – Whether to use `abl_cache` for previously abduced candidates to speed up subsequent operations. Defaults to `True`.
- **key_func** (*Callable*, *optional*) – A function employed for hashing in `abl_cache`. This is only operational when `use_cache` is set to `True`. Defaults to `to_hashable`.
- **cache_size** (*int*, *optional*) – The cache size in `abl_cache`. This is only operational when `use_cache` is set to `True`. Defaults to 4096.

Notes

Users should derive from this base class to build their own knowledge base. For the user-build KB (a derived subclass), it's only required for the user to provide the `pseudo_label_list` and override the `logic_forward` function (specifying how to perform logical reasoning). After that, other operations (e.g. how to perform abductive reasoning) will be automatically set up.

abduce_candidates(*pseudo_label*: List[Any], *y*: Any, *x*: List[Any], *max_revision_num*: int, *require_more_revision*: int) → List[List[Any]]

Perform abductive reasoning to get a candidate compatible with the knowledge base.

Parameters

- **pseudo_label** (List[Any]) – Pseudo-labels of an example (to be revised by abductive reasoning).
- **y** (Any) – Ground truth of the reasoning result for the example.
- **x** (List[Any]) – The example. If the information from the example is not required in the reasoning process, then this parameter will not have any effect.
- **max_revision_num** (int) – The upper limit on the number of revised labels for each example.
- **require_more_revision** (int) – Specifies additional number of revisions permitted beyond the minimum required.

Returns

A tuple of two elements. The first element is a list of candidate revisions, i.e. revised pseudo-labels of the example. that are compatible with the knowledge base. The second element is a list of reasoning results corresponding to each candidate, i.e., the outcome of the `logic_forward` function.

Return type

Tuple[List[List[Any]], List[Any]]

abstract logic_forward(*pseudo_label*: List[Any], *x*: Optional[List[Any]] = None) → Any

How to perform (deductive) logical reasoning, i.e. matching an example's pseudo-labels to its reasoning result. Users are required to provide this.

Parameters

- **pseudo_label** (List[Any]) – Pseudo-labels of an example.
- **x** (List[Any], optional) – The example. If deductive logical reasoning does not require any information from the example, the overridden function provided by the user can omit this parameter.

Returns

The reasoning result.

Return type

Any

revise_at_idx(*pseudo_label*: List[Any], *y*: Any, *x*: List[Any], *revision_idx*: List[int]) → List[List[Any]]

Revise the pseudo-labels at specified index positions.

Parameters

- **pseudo_label** (List[Any]) – Pseudo-labels of an example (to be revised).
- **y** (Any) – Ground truth of the reasoning result for the example.

- **x** (*List[Any]*) – The example. If the information from the example is not required in the reasoning process, then this parameter will not have any effect.
- **revision_idx** (*List[int]*) – A list specifying indices of where revisions should be made to the pseudo-labels.

Returns

A tuple of two elements. The first element is a list of candidate revisions, i.e. revised pseudo-labels of the example. that are compatible with the knowledge base. The second element is a list of reasoning results corresponding to each candidate, i.e., the outcome of the `logic_forward` function.

Return type

Tuple[List[List[Any]], List[Any]]

class `ablkit.reasoning.PrologKB`(*pseudo_label_list: List[Any], pl_file: str*)

Bases: *KBBase*

Knowledge base provided by a Prolog (.pl) file.

Parameters

- **pseudo_label_list** (*List[Any]*) – Refer to class *KBBase*.
- **pl_file** (*str*) – Prolog file containing the KB.

Notes

Users can instantiate this class to build their own knowledge base. During the instantiation, users are only required to provide the `pseudo_label_list` and `pl_file`. To use the default logic forward and abductive reasoning methods in this class, in the Prolog (.pl) file, there needs to be a rule which is strictly formatted as `logic_forward(Pseudo_labels, Res)`, e.g., `logic_forward([A,B], C) :- C is A+B`. For specifics, refer to the `logic_forward` and `get_query_string` functions in this class. Users are also welcome to override related functions for more flexible support.

get_query_string(*pseudo_label: List[Any], y: Any, x: List[Any], revision_idx: List[int]*) → *str*

Get the query to be used for consulting Prolog. This is a default function for demo, users would override this function to adapt to their own Prolog file. In this demo function, return query `logic_forward([kept_labels, Revise_labels], Res)`.

Parameters

- **pseudo_label** (*List[Any]*) – Pseudo-labels of an example (to be revised by abductive reasoning).
- **y** (*Any*) – Ground truth of the reasoning result for the example.
- **x** (*List[Any]*) – The corresponding input example. If the information from the input is not required in the reasoning process, then this parameter will not have any effect.
- **revision_idx** (*List[int]*) – A list specifying indices of where revisions should be made to the pseudo-labels.

Returns

A string of the query.

Return type

str

logic_forward(*pseudo_label*: List[Any], *x*: Optional[List[Any]] = None) → Any

Consult prolog with the query `logic_forward(pseudo_labels, Res)`., and set the returned `Res` as the reasoning results. To use this default function, there must be a `logic_forward` method in the `pl` file to perform reasoning. Otherwise, users would override this function.

Parameters

- **pseudo_label** (List[Any]) – Pseudo-labels of an example.
- **x** (List[Any]) – The corresponding input example. If the information from the input is not required in the reasoning process, then this parameter will not have any effect.

revise_at_idx(*pseudo_label*: List[Any], *y*: Any, *x*: List[Any], *revision_idx*: List[int]) → List[List[Any]]

Revise the pseudo-labels at specified index positions by querying Prolog.

Parameters

- **pseudo_label** (List[Any]) – Pseudo-labels of an example (to be revised).
- **y** (Any) – Ground truth of the reasoning result for the example.
- **x** (List[Any]) – The corresponding input example. If the information from the input is not required in the reasoning process, then this parameter will not have any effect.
- **revision_idx** (List[int]) – A list specifying indices of where revisions should be made to the pseudo-labels.

Returns

A tuple of two elements. The first element is a list of candidate revisions, i.e. revised pseudo-labels of the example. that are compatible with the knowledge base. The second element is a list of reasoning results corresponding to each candidate, i.e., the outcome of the `logic_forward` function.

Return type

Tuple[List[List[Any]], List[Any]]

class `ablkit.reasoning.Reasoner`(*kb*: KBBBase, *dist_func*: Union[str, Callable] = 'confidence', *idx_to_label*: Optional[dict] = None, *max_revision*: Union[int, float] = -1, *require_more_revision*: int = 0, *use_zoopt*: bool = False)

Bases: object

Reasoner for minimizing the inconsistency between the knowledge base and learning models.

Parameters

- **kb** (class KBBBase) – The knowledge base to be used for reasoning.
- **dist_func** (Union[str, Callable], optional) – The distance function used to determine the cost list between each candidate and the given prediction. The cost is also referred to as a consistency measure, wherein the candidate with lowest cost is selected as the final abduced label. It can be either a string representing a predefined distance function or a callable function. The available predefined distance functions: 'hamming' | 'confidence' | 'avg_confidence'. 'hamming' directly calculates the Hamming distance between the predicted pseudo-label in the data example and each candidate. 'confidence' and 'avg_confidence' calculates the confidence distance between the predicted probabilities in the data example and each candidate, where the confidence distance is defined as 1 - the product of prediction probabilities in 'confidence' and 1 - the average of prediction probabilities in 'avg_confidence'. Alternatively, the callable function should have the signature `dist_func(data_example, candidates, candidate_idxes, reasoning_results)` and must return a cost list. Each element in

this cost list should be a numerical value representing the cost for each candidate, and the list should have the same length as candidates. Defaults to 'confidence'.

- **idx_to_label** (*dict*, *optional*) – A mapping from index in the base model to label. If not provided, a default order-based index to label mapping is created. Defaults to None.
- **max_revision** (*Union[int, float]*, *optional*) – The upper limit on the number of revisions for each data example when performing abductive reasoning. If float, denotes the fraction of the total length that can be revised. A value of -1 implies no restriction on the number of revisions. Defaults to -1.
- **require_more_revision** (*int*, *optional*) – Specifies additional number of revisions permitted beyond the minimum required when performing abductive reasoning. Defaults to 0.
- **use_zoopt** (*bool*, *optional*) – Whether to use ZOOpt library during abductive reasoning. Defaults to False.

abduce(*data_example*: [ListData](#)) → List[Any]

Perform abductive reasoning on the given data example.

Parameters

data_example ([ListData](#)) – Data example.

Returns

A revised pseudo-labels of the example through abductive reasoning, which is compatible with the knowledge base.

Return type

List[Any]

batch_abduce(*data_examples*: [ListData](#)) → List[List[Any]]

Perform abductive reasoning on the given prediction data examples. For detailed information, refer to [abduce](#).

zoopt_budget(*symbol_num*: *int*) → *int*

Set the budget for ZOOpt optimization. The budget can be dynamic relying on the number of symbols considered, e.g., the default implementation shown below. Alternatively, it can be a fixed value, such as simply setting it to 100.

Parameters

symbol_num (*int*) – The number of symbols to be considered in the ZOOpt optimization process.

Returns

The budget for ZOOpt optimization.

Return type

int

zoopt_score(*symbol_num*: *int*, *data_example*: [ListData](#), *sol*: [Solution](#)) → *int*

Set the score for a solution. A lower score suggests that ZOOpt library has a higher preference for this solution.

Parameters

- **symbol_num** (*int*) – Number of total symbols.
- **data_example** ([ListData](#)) – Data example.
- **sol** ([Solution](#)) – The solution for ZOOpt library.

Returns

The score for the solution.

Return type

int

2.17 ablkit.bridge

class `ablkit.bridge.BaseBridge`(*model*: [ABLModel](#), *reasoner*: [Reasoner](#))

Bases: `object`

A base class for bridging learning and reasoning parts.

This class provides necessary methods that need to be overridden in subclasses to construct a typical pipeline of Abductive Learning (corresponding to `train`), which involves the following four methods:

- `predict`: Predict class indices on the given data examples.
- `idx_to_pseudo_label`: Map indices into pseudo-labels.
- `abduce_pseudo_label`: Revise pseudo-labels based on abductive reasoning.
- `pseudo_label_to_idx`: Map revised pseudo-labels back into indices.

Parameters

- **model** ([ABLModel](#)) – The machine learning model wrapped in [ABLModel](#), which is mainly used for prediction and model training.
- **reasoner** ([Reasoner](#)) – The reasoning part wrapped in [Reasoner](#), which is used for pseudo-label revision.

abstract `abduce_pseudo_label`(*data_examples*: [ListData](#)) → List[List[Any]]

Placeholder for revising pseudo-labels based on abductive reasoning.

filter_pseudo_label(*data_examples*: [ListData](#)) → List[List[Any]]

Default filter function for pseudo-label.

abstract `idx_to_pseudo_label`(*data_examples*: [ListData](#)) → List[List[Any]]

Placeholder for mapping indices to pseudo-labels.

abstract `predict`(*data_examples*: [ListData](#)) → Tuple[List[List[Any]], List[List[Any]]]

Placeholder for predicting class indices from input.

abstract `pseudo_label_to_idx`(*data_examples*: [ListData](#)) → List[List[Any]]

Placeholder for mapping pseudo-labels to indices.

abstract `test`(*test_data*: Union[[ListData](#), Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]])
→ None

Placeholder for model validation.

abstract `train`(*train_data*: Union[[ListData](#), Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]])

Placeholder for training loop of Abductive Learning.

abstract `valid`(*val_data*: Union[[ListData](#), Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]])
→ None

Placeholder for model test.

class ablkit.bridge.**SimpleBridge**(*model*: ABLModel, *reasoner*: Reasoner, *metric_list*: List[BaseMetric])

Bases: *BaseBridge*

A basic implementation for bridging machine learning and reasoning parts.

This class implements the typical pipeline of Abductive Learning, which involves the following five steps:

- Predict class probabilities and indices for the given data examples.
- Map indices into pseudo-labels.
- Revise pseudo-labels based on abductive reasoning.
- Map the revised pseudo-labels to indices.
- Train the model.

Parameters

- **model** (ABLModel) – The machine learning model wrapped in ABLModel, which is mainly used for prediction and model training.
- **reasoner** (Reasoner) – The reasoning part wrapped in Reasoner, which is used for pseudo-label revision.
- **metric_list** (List[BaseMetric]) – A list of metrics used for evaluating the model's performance.

abduce_pseudo_label(*data_examples*: ListData) → List[List[Any]]

Revise predicted pseudo-labels of the given data examples using abduction.

Parameters

data_examples (ListData) – Data examples containing predicted pseudo-labels.

Returns

A list of abduced pseudo-labels for the given data examples.

Return type

List[List[Any]]

concat_data_examples(*unlabel_data_examples*: ListData, *label_data_examples*: Optional[ListData]) → ListData

Concatenate unlabeled and labeled data examples. `abduced_pseudo_label` of unlabeled data examples and `gt_pseudo_label` of labeled data examples will be used to train the model.

Parameters

- **unlabel_data_examples** (ListData) – Unlabeled data examples to concatenate.
- **label_data_examples** (ListData, optional) – Labeled data examples to concatenate, if available.

Returns

Concatenated data examples.

Return type

ListData

data_preprocess(*prefix*: str, *data*: Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]]) → ListData

Transform data in the form of (X, gt_pseudo_label, Y) into ListData.

Parameters

- **prefix** (*str*) – A prefix indicating the type of data processing (e.g., ‘train’, ‘test’).
- **data** (*Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]]*) – Data to be preprocessed. Can be ListData or a tuple of lists.

Returns

The preprocessed ListData object.

Return type

ListData

idx_to_pseudo_label (*data_examples: ListData*) → List[List[Any]]

Map indices of data examples into pseudo-labels.

Parameters

data_examples (*ListData*) – Data examples containing the indices.

Returns

A list of pseudo-labels converted from indices.

Return type

List[List[Any]]

predict (*data_examples: ListData*) → Tuple[List[ndarray], List[ndarray]]

Predict class indices and probabilities (if `predict_proba` is implemented in `self.model.base_model`) on the given data examples.

Parameters

data_examples (*ListData*) – Data examples on which predictions are to be made.

Returns

A tuple containing lists of predicted indices and probabilities.

Return type

Tuple[List[ndarray], List[ndarray]]

pseudo_label_to_idx (*data_examples: ListData*) → List[List[Any]]

Map pseudo-labels of data examples into indices.

Parameters

data_examples (*ListData*) – Data examples containing pseudo-labels.

Returns

A list of indices converted from pseudo-labels.

Return type

List[List[Any]]

test (*test_data: Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], Optional[List[Any]]]*) → None

Test the model with the given test data.

Parameters

test_data (*Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], Optional[List[Any]]]*) – Test data should be in the form of (X, gt_pseudo_label, Y) or a ListData object with X, gt_pseudo_label and Y attributes. Both gt_pseudo_label and Y can be either None or not, which depends on the evaluation metrics in `self.metric_list`.

```
train(train_data: Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]]],
      label_data: Optional[Union[ListData, Tuple[List[List[Any]], List[List[Any]], List[Any]]] = None,
      val_data: Optional[Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]],
      Optional[List[Any]]]] = None, loops: int = 50, segment_size: Union[int, float] = 1.0, eval_interval:
      int = 1, save_interval: Optional[int] = None, save_dir: Optional[str] = None)
```

A typical training pipeline of Abuductive Learning.

Parameters

- **train_data** (Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], List[Any]]) – Training data should be in the form of (X, gt_pseudo_label, Y) or a ListData object with X, gt_pseudo_label and Y attributes. - X is a list of sublists representing the input data. - gt_pseudo_label is only used to evaluate the performance of the ABLModel but not to train. gt_pseudo_label can be None. - Y is a list representing the ground truth reasoning result for each sublist in X.
- **label_data** (Union[ListData, Tuple[List[List[Any]], List[List[Any]], List[Any]]], optional) – Labeled data should be in the same format as train_data. The only difference is that the gt_pseudo_label in label_data should not be None and will be utilized to train the model. Defaults to None.
- **val_data** (Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], Optional[List[Any]]], optional) – Validation data should be in the same format as train_data. Both gt_pseudo_label and Y can be either None or not, which depends on the evaluation metrics in self.metric_list. If val_data is None, train_data will be used to validate the model during training time. Defaults to None.
- **loops** (int) – Learning part and Reasoning part will be iteratively optimized for loops times. Defaults to 50.
- **segment_size** (Union[int, float]) – Data will be split into segments of this size and data in each segment will be used together to train the model. Defaults to 1.0.
- **eval_interval** (int) – The model will be evaluated every eval_interval loop during training. Defaults to 1.
- **save_interval** (int, optional) – The model will be saved every eval_interval loop during training. Defaults to None.
- **save_dir** (str, optional) – Directory to save the model. Defaults to None.

```
valid(val_data: Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], Optional[List[Any]]])
→ None
```

Validate the model with the given validation data.

Parameters

- **val_data** (Union[ListData, Tuple[List[List[Any]], Optional[List[List[Any]]], Optional[List[Any]]]) – Validation data should be in the form of (X, gt_pseudo_label, Y) or a ListData object with X, gt_pseudo_label and Y attributes. Both gt_pseudo_label and Y can be either None or not, which depends on the evaluation metrics in self.metric_list.

2.18 ablkit.utils

class `ablkit.utils.ABLLogger`(*name*: str, *logger_name*='abl', *log_file*: Optional[str] = None, *log_level*: Union[int, str] = 'INFO', *file_mode*: str = 'w')

Bases: `Logger`, `ManagerMixin`

Formatted logger used to record messages with different log levels and features.

`ABLLogger` provides a formatted logger that can log messages with different log levels. It allows the creation of logger instances in a similar manner to `ManagerMixin`. The logger has features like distributed log storage and colored terminal output for different log levels.

Parameters

- **name** (str) – Global instance name.
- **logger_name** (str, optional) – name attribute of `logging.Logger` instance. Defaults to 'abl'.
- **log_file** (str, optional) – The log filename. If specified, a `FileHandler` will be added to the logger. Defaults to None.
- **log_level** (Union[int, str], optional) – The log level of the handler. Defaults to 'INFO'. If log level is 'DEBUG', distributed logs will be saved during distributed training.
- **file_mode** (str, optional) – The file mode used to open log file. Defaults to 'w'.

Notes

- The name of the logger and the `instance_name` of `ABLLogger` could be different. `ABLLogger` instances are retrieved using `ABLLogger.get_instance`, not `logging.getLogger`. This ensures `ABLLogger` is not influenced by third-party logging configurations.
- Unlike `logging.Logger`, `ABLLogger` will not log warning or error messages without `Handler`.

Examples

```
>>> logger = ABLLogger.get_instance(name='ABLLogger', logger_name='Logger')
>>> # Although logger has a name attribute like `logging.Logger`
>>> # We cannot get logger instance by `logging.getLogger`.
>>> assert logger.name == 'Logger'
>>> assert logger.instance_name == 'ABLLogger'
>>> assert id(logger) != id(logging.getLogger('Logger'))
>>> # Get logger that does not store logs.
>>> logger1 = ABLLogger.get_instance('logger1')
>>> # Get logger only save rank0 logs.
>>> logger2 = ABLLogger.get_instance('logger2', log_file='out.log')
>>> # Get logger only save multiple ranks logs.
>>> logger3 = ABLLogger.get_instance('logger3', log_file='out.log',
↪distributed=True)
```

callHandlers(*record*: `LogRecord`) → None

Pass a record to all relevant handlers.

Override the `callHandlers` method in `logging.Logger` to avoid multiple warning messages in DDP mode. This method loops through all handlers of the logger instance and its parents in the logger hierarchy.

Parameters

record (*LogRecord*) – A *LogRecord* instance containing the logged message.

classmethod **get_current_instance()** → *ABLLogger*

Get the latest created *ABLLogger* instance.

Returns

The latest created *ABLLogger* instance. If no instance has been created, returns a logger with the instance name “abl”.

Return type

ABLLogger

property **log_dir**

Get the directory where the log is stored.

Returns

Directory where the log is stored.

Return type

str

property **log_file**

Get the file path of the log.

Returns

Path of the log.

Return type

str

setLevel(*level*)

Set the logging level of this logger.

Override the **setLevel** method to clear caches of all *ABLLogger* instances managed by *ManagerMixin*. The level must be an int or a str.

Parameters

level (*Union[int, str]*) – The logging level to set.

class **ablkit.utils.Cache**(*func: Callable[[K], T]*)

Bases: *Generic[K, T]*

A generic caching mechanism that stores the results of a function call and retrieves them to avoid repeated calculations.

This class implements a dictionary-based cache with a circular doubly linked list to manage the cache entries efficiently. It is designed to be generic, allowing for caching of any callable function.

Parameters

func (*Callable[[K], T]*) – The function to be cached. This function takes an argument of type *K* and returns a value of type *T*.

clear_cache()

Invalidate the entire cache.

get_from_dict(*obj, *args*) → *T*

Retrieve a value from the cache or compute it using **self.func**.

Parameters

- **obj** (*Any*) – The object to which the cached method/function belongs.

- ***args** (*Any*) – Arguments used in key generation for cache retrieval or function computation.

Returns

The value from the cache or computed by the function.

Return type

T

init_cache(*obj*)

Initialize the cache settings.

Parameters

obj (*Any*) – The object containing settings for cache initialization.

ablkit.utils.abl_cache()

Decorator to enable caching for a function.

Returns

The wrapped function with caching capability.

Return type

Callable

ablkit.utils.avg_confidence_dist(*pred_prob: ndarray, candidates_idx: List[List[Any]]*) → ndarray

Compute the average confidence distance between prediction probabilities and candidates, where the confidence distance is defined as 1 - the average of prediction probabilities.

Parameters

- **pred_prob** (*np.ndarray*) – Prediction probability distributions, each element is an array representing the probability distribution of a particular prediction.
- **candidates_idx** (*List[List[Any]]*) – Multiple possible candidates' indices.

Returns

Confidence distances computed for each candidate.

Return type

np.ndarray

ablkit.utils.confidence_dist(*pred_prob: ndarray, candidates_idx: List[List[Any]]*) → ndarray

Compute the confidence distance between prediction probabilities and candidates, where the confidence distance is defined as 1 - the product of prediction probabilities.

Parameters

- **pred_prob** (*np.ndarray*) – Prediction probability distributions, each element is an array representing the probability distribution of a particular prediction.
- **candidates_idx** (*List[List[Any]]*) – Multiple possible candidates' indices.

Returns

Confidence distances computed for each candidate.

Return type

np.ndarray

ablkit.utils.flatten(*nested_list: List[Union[Any, List[Any], Tuple[Any, ...]]]*) → List[Any]

Flattens a nested list at the first level.

Parameters

nested_list (*List[Union[Any, List[Any], Tuple[Any, ...]]*) – A list which might contain sublists or tuples at the first level.

Returns

A flattened version of the input list, where only the first level of sublists and tuples are reduced.

Return type

List[Any]

ablkit.utils.**hamming_dist**(*pred_pseudo_label: List[Any], candidates: List[List[Any]]*) → ndarray

Compute the Hamming distance between two arrays.

Parameters

- **pred_pseudo_label** (*List[Any]*) – Pseudo-labels of an example.
- **candidates** (*List[List[Any]]*) – Multiple possible candidates.

Returns

Hamming distances computed for each candidate.

Return type

np.ndarray

ablkit.utils.**print_log**(*msg, logger: Optional[Union[Logger, str]] = None, level: Optional[int] = 20*) → None

Print a log message using the specified logger or a default method.

This function logs a message with a given logger, if provided, or prints it using the standard `print` function. It supports special logger types such as ‘silent’ and ‘current’.

Parameters

- **msg** (*str*) – The message to be logged.
- **logger** (*Union[Logger, str], optional*) – The logger to use for logging the message. It can be a `logging.Logger` instance, a string specifying the logger name, ‘silent’, ‘current’, or None. If None, the `print` method is used. - ‘silent’: No message will be printed. - ‘current’: Use the latest created logger to log the message. - other str: The instance name of the logger. A `ValueError` is raised if the logger has not been created. - None: The `print()` method is used for logging.
- **level** (*int, optional*) – The logging level. This is only applicable when `logger` is a `Logger` object, ‘current’, or a named logger instance. The default is `logging.INFO`.

ablkit.utils.**reform_list**(*flattened_list: List[Any], structured_list: List[Union[Any, List[Any], Tuple[Any, ...]]]*) → List[List[Any]]

Reform the list based on the structure of `structured_list`.

Parameters

- **flattened_list** (*List[Any]*) – A flattened list of elements.
- **structured_list** (*List[Union[Any, List[Any], Tuple[Any, ...]]]*) – A list that reflects the desired structure, which may contain sublists or tuples.

Returns

A reformed list that mimics the structure of `structured_list`.

Return type

List[List[Any]]

`ablkit.utils.tab_data_to_tuple(X: Union[List[Any], Any], y: Union[List[Any], Any], reasoning_result: Optional[Any] = 0) → Tuple[List[List[Any]], List[List[Any]], List[Any]]`

Convert a tabular data to a tuple by adding a dimension to each element of X and y. The tuple contains three elements: data, label, and reasoning result. If X is None, return None.

Parameters

- **X** (*Union[List[Any], Any]*) – The data.
- **y** (*Union[List[Any], Any]*) – The label.
- **reasoning_result** (*Any, optional*) – The reasoning result. Defaults to 0.

Returns

A tuple of (data, label, reasoning_result).

Return type

`Tuple[List[List[Any]], List[List[Any]], List[Any]]`

`ablkit.utils.to_hashable(x: Union[List[Any], Any]) → Union[Tuple[Any, ...], Any]`

Convert a nested list to a nested tuple so it is hashable.

Parameters

x (*Union[List[Any], Any]*) – A potentially nested list to convert to a tuple.

Returns

The input converted to a tuple if it was a list, otherwise the original input.

Return type

`Union[Tuple[Any, ...], Any]`

2.19 References

Zhi-Hua Zhou. [Abductive learning: Towards bridging machine learning and logical reasoning](#). **Science China Information Sciences**, 2019, 62: 076101.

Zhi-Hua Zhou and Yu-Xuan Huang. [Abductive learning](#). In P. Hitzler and M. K. Sarker eds., **Neuro-Symbolic Artificial Intelligence: The State of the Art**, IOP Press, Amsterdam, 2022, p.353-379

```
@article{zhou2019abductive,
  title      = {Abductive learning: towards bridging machine learning and logical_
↪reasoning},
  author     = {Zhou, Zhi-Hua},
  journal    = {Science China Information Sciences},
  volume     = {62},
  number     = {7},
  pages      = {76101},
  year       = {2019}
}

@incollection{zhou2022abductive,
  title      = {Abductive Learning},
  author     = {Zhou, Zhi-Hua and Huang, Yu-Xuan},
  booktitle  = {Neuro-Symbolic Artificial Intelligence: The State of the Art},
  editor     = {Pascal Hitzler and Md. Kamruzzaman Sarker},
  publisher  = {{IOS} Press},
  pages      = {353--369},
}
```

(continues on next page)

(continued from previous page)

```
    address    = {Amsterdam},  
    year       = {2022}  
}
```


PYTHON MODULE INDEX

a

`ablkit.bridge`, [59](#)
`ablkit.data.evaluation`, [47](#)
`ablkit.learning.torch_dataset`, [53](#)
`ablkit.reasoning`, [53](#)
`ablkit.utils`, [63](#)

A

`abduce()` (*abltk.reasoning.Reasoner* method), 58
`abduce_candidates()` (*abltk.reasoning.GroundKB* method), 54
`abduce_candidates()` (*abltk.reasoning.KBBase* method), 55
`abduce_pseudo_label()` (*abltk.bridge.BaseBridge* method), 59
`abduce_pseudo_label()` (*abltk.bridge.SimpleBridge* method), 60
`abl_cache()` (in module *abltk.utils*), 65
`abltk.bridge`
 module, 59
`abltk.data.evaluation`
 module, 47
`abltk.learning.torch_dataset`
 module, 53
`abltk.reasoning`
 module, 53
`abltk.utils`
 module, 63
`ABLLogger` (class in *abltk.utils*), 63
`ABLModel` (class in *abltk.learning*), 49
`avg_confidence_dist()` (in module *abltk.utils*), 65

B

`BaseBridge` (class in *abltk.bridge*), 59
`BaseMetric` (class in *abltk.data.evaluation*), 47
`BasicNN` (class in *abltk.learning*), 50
`batch_abduce()` (*abltk.reasoning.Reasoner* method), 58

C

`Cache` (class in *abltk.utils*), 64
`callHandlers()` (*abltk.utils.ABLLogger* method), 63
`ClassificationDataset` (class in *abltk.learning.torch_dataset*), 53
`clear_cache()` (*abltk.utils.Cache* method), 64
`compute_metrics()` (*abltk.data.evaluation.BaseMetric* method), 47
`compute_metrics()` (*abltk.data.evaluation.ReasoningMetric* method), 48

`compute_metrics()` (*abltk.data.evaluation.SymbolAccuracy* method), 48
`concat_data_examples()` (*abltk.bridge.SimpleBridge* method), 60
`confidence_dist()` (in module *abltk.utils*), 65

D

`data_preprocess()` (*abltk.bridge.SimpleBridge* method), 60

E

`elements_num()` (*abltk.data.structures.ListData* method), 46
`evaluate()` (*abltk.data.evaluation.BaseMetric* method), 47

F

`filter_pseudo_label()` (*abltk.bridge.BaseBridge* method), 59
`fit()` (*abltk.learning.BasicNN* method), 50
`flatten()` (*abltk.data.structures.ListData* method), 46
`flatten()` (in module *abltk.utils*), 65

G

`get_current_instance()` (*abltk.utils.ABLLogger* class method), 64
`get_from_dict()` (*abltk.utils.Cache* method), 64
`get_query_string()` (*abltk.reasoning.PrologKB* method), 56
`GroundKB` (class in *abltk.reasoning*), 53

H

`hamming_dist()` (in module *abltk.utils*), 66

I

`idx_to_pseudo_label()` (*abltk.bridge.BaseBridge* method), 59
`idx_to_pseudo_label()` (*abltk.bridge.SimpleBridge* method), 61
`init_cache()` (*abltk.utils.Cache* method), 65

K

KBBase (class in *abltk.reasoning*), 54

L

ListData (class in *abltk.data.structures*), 45
 load() (*abltk.learning.ABLModel* method), 49
 load() (*abltk.learning.BasicNN* method), 51
 log_dir (*abltk.utils.ABLLogger* property), 64
 log_file (*abltk.utils.ABLLogger* property), 64
 logic_forward() (*abltk.reasoning.KBBase* method), 55
 logic_forward() (*abltk.reasoning.PrologKB* method), 56

M

module
 abltk.bridge, 59
 abltk.data.evaluation, 47
 abltk.learning.torch_dataset, 53
 abltk.reasoning, 53
 abltk.utils, 63

P

predict() (*abltk.bridge.BaseBridge* method), 59
 predict() (*abltk.bridge.SimpleBridge* method), 61
 predict() (*abltk.learning.ABLModel* method), 49
 predict() (*abltk.learning.BasicNN* method), 51
 predict_proba() (*abltk.learning.BasicNN* method), 51
 PredictionDataset (class in *abltk.learning.torch_dataset*), 53
 print_log() (in module *abltk.utils*), 66
 process() (*abltk.data.evaluation.BaseMetric* method), 47
 process() (*abltk.data.evaluation.ReasoningMetric* method), 48
 process() (*abltk.data.evaluation.SymbolAccuracy* method), 48
 PrologKB (class in *abltk.reasoning*), 56
 pseudo_label_to_idx() (*abltk.bridge.BaseBridge* method), 59
 pseudo_label_to_idx() (*abltk.bridge.SimpleBridge* method), 61

R

Reasoner (class in *abltk.reasoning*), 57
 ReasoningMetric (class in *abltk.data.evaluation*), 47
 reform_list() (in module *abltk.utils*), 66
 RegressionDataset (class in *abltk.learning.torch_dataset*), 53
 revise_at_idx() (*abltk.reasoning.KBBase* method), 55
 revise_at_idx() (*abltk.reasoning.PrologKB* method), 57

S

save() (*abltk.learning.ABLModel* method), 49
 save() (*abltk.learning.BasicNN* method), 52
 score() (*abltk.learning.BasicNN* method), 52
 setLevel() (*abltk.utils.ABLLogger* method), 64
 SimpleBridge (class in *abltk.bridge*), 59
 SymbolAccuracy (class in *abltk.data.evaluation*), 48

T

tab_data_to_tuple() (in module *abltk.utils*), 66
 test() (*abltk.bridge.BaseBridge* method), 59
 test() (*abltk.bridge.SimpleBridge* method), 61
 to_hashable() (in module *abltk.utils*), 67
 to_tuple() (*abltk.data.structures.ListData* method), 46
 train() (*abltk.bridge.BaseBridge* method), 59
 train() (*abltk.bridge.SimpleBridge* method), 61
 train() (*abltk.learning.ABLModel* method), 49
 train_epoch() (*abltk.learning.BasicNN* method), 52

V

valid() (*abltk.bridge.BaseBridge* method), 59
 valid() (*abltk.bridge.SimpleBridge* method), 62
 valid() (*abltk.learning.ABLModel* method), 49

Z

zoopt_budget() (*abltk.reasoning.Reasoner* method), 58
 zoopt_score() (*abltk.reasoning.Reasoner* method), 58