errudite

Release 0.0.1

Tongshuang Wu

May 06, 2020

CONTENTS

Errudite is an opensourced software tool for performing and sharing error analyses in natural language processing tasks. Errudite is designed following three principles:

- First, error groups should be precisely defined for reproducibility; Errudite supports this with an expressive domain- specific language.
- Second, to avoid spurious conclusions, a large set of instances should be analyzed, including both positive and neg- ative examples; Errudite enables systematic grouping of relevant instances with filtering queries.
- Third, hypotheses about the cause of errors should be explicitly tested; Errudite supports this via automated counterfactual rewriting.

CHAPTER

ONE

ABSTRACT DATA CLASSES

As the basis for extending Errudite to other tasks and predictors, we use two abstract classes to:

- `register` the customized classes or functions via `Registrable`, so we could add our own code without extensively touching the source folder.
- `store` the objects in a hash variable via `Store`, so all the created instances and analyses can be easily queried and used in various functions.

1.1 Abstract Data Classes

As the basis of the implementation, we use the following two abstract classes:

1.1.1 errudite.utils.registrable

`Registrable` is inspired by allennlp.common.registrable. It is a "mixin" classes for endowing base class with a named registry for its subclasses and a decorator for registering them.

class `errudite.utils.registrable.`**Registrable**

Adjusted from https://allenai.github.io/allennlp-docs/api/allennlp.common.registrable.html. Any class that inherits from `Registrable` gains access to a named registry for its subclasses. To register them, just decorate them with the classmethod `@BaseClass.register(name)`. After which you can call `BaseClass.list_available()` to get the keys for the registered subclasses, and `BaseClass.by_name(name)` to get the corresponding subclass. Note that the registry stores the subclasses themselves; not class instances.

Note that if you use this class to implement a new `Registrable` abstract class, you must ensure that all subclasses of the abstract class are loaded when the module is loaded, because the subclasses register themselves in their respective files. You can achieve this by having the abstract class and all subclasses in the `__init__.py` of the module in which they reside (as this causes any import of either the abstract class or a subclass to load all other subclasses and the abstract class).

classmethod by_name(*name: str*) → Type[T]

Get the sub-class/sub-function by their registered name.

Parameters

- **cls** (*Type[T]*) – The base class.
- **name** (*str*) – The name of the subclass/sub-function.

Returns sub-class/sub-function

Return type Type[T]

classmethod list_available() → List[str]
List all the available sub-class/functions available in an abstracted class.

Returns The string list of all the sub-class/and sub-functions.

Return type List[str]

classmethod register(*name: str = None*) → Callable
A decorator function that helps register sub-classes/sub-functions: `@BaseClass.register(name)`

Parameters

- **cls** (*Type[T]*) – The base class
- **name** (*str, optional*) – The name of the subclass. If not given retrive the name of the subclasses/functions. By default None

Returns The registering function

Return type Callable

1.1.2 errudite.utils.store

`Store` is a storage class that wraps the same types of objects. It provides functions that assist imports from and exports to `.json` files, and functions that make the Store class work just like a `dict` object.

class errudite.utils.store.**Store**
A store wrapper class. Children of this class allow actual objects to be saved in `Store._store_hash[cls] = { object_key: object }`. We provide functions that helpes the Store class to be used mostly like a normal `dict` object.

classmethod create_from_json(*raw: Dict[str, Any]*) → T
Recreate the object from its seralized raw json file.

Parameters **raw** (*Dict[str, Any]*) – The raw version of the object.

Returns The re-created object.

Return type T

Raises **NotImplementedError** – This function is supposed to be implemented

classmethod exists(*name: str*) → bool
Check with the stored object exist, by querying its name.

Parameters **name** (*str*) – The name of the intended stored object.

Returns If the instance exists.

Return type bool

classmethod export_to_file(*file_name: str*) → bool
Export the store hash `Store._store_hash[cls]` to json file.

Parameters **file_name** (*str*) – The name of the json file. It will save to a file: `{CACHE_FOLDERS["analysis"]}/{file_name}.json`.

Returns Whether or not the export is successful.

Return type bool

classmethod get(*name: str*) → T
Get the stored object by querying its name.

Parameters **name** (*str*) – The name of the intended stored object.

Returns The queried object.

Return type T

classmethod import_from_file(*file_name: str*) → Dict[str, T]
Import the saved store frome a file. It recovers all the saved, json version of objects, and save them to the store_hash.

Parameters **file_name** (*str*) – The name of the json file. It should be a file in `CACHE_FOLDERS["analysis"]`.

Returns The restored hash, or `Store._store_hash[cls]`.

Return type Dict[str, T]

classmethod items() → Iterable[Tuple[str, T]]
Return the items of store hash as a generator of tuples of `(key, val)`.

Returns The generator of the tuples of keys and values.

Return type Iterable[Tuple[str, T]]

classmethod keys() → Iterable[str]
Return the keys of the store hash as a generator.

Returns The generator of the keys

Return type Iterable[str]

classmethod remove_saved(*name: str*) → bool
Remove the object from the store hash by querying its name.

Parameters **name** (*str*) – The name of the intended stored object.

Returns If the removal is successful.

Return type bool

classmethod save(*obj: T*) → bool
Save the object to the store hash.

Parameters **obj** (*T*) – The object to be saved.

Returns Whether or not the export is successful.

Return type bool

classmethod store_hash() → Dict[str, T]
Return the store hash as a dict.

Returns

Return type Dict[str, T]

classmethod values() → Iterable[T]
Return the values of the store hash as a generator.

Returns The generator of the values

Return type Iterable[T]

CHAPTER

TWO

DATA STRUCTURE DESIGN

Regardless of the task, raw data are transferred into `Instance` s, which in turn are homogeneous collections of `Target` s. Which Targets they contain depends on the type of data. For example, for machine comprehension (or datasets like SQAuD), an instance will have `Question`, `Context`, `Groundtruths`, and `Predictions`.

2.1 Targets and Instances

2.1.1 errudite.targets.target

Targets are primitives which allow users to access inputs and outputs at different levels of granularity, such as the `question (q)`, passage `context (c)`, `ground truth (g)`, the prediction of a model m (denoted by `prediction(model="m")`), sentence and token. Targets can be composed, e.g., `sentence(g)` extracts the sentence that contains the ground truth span.

class errudite.targets.target.**Target**(*qid: str*, *text: str*, *vid: int = 0*, *annotator: errudite.processor.spacy_annotator.SpacyAnnotator = None*, *metas: Dict[str, any] = {}*)

A batch of Instances. In addition to containing the instances themselves, it contains helper functions for converting the data into tensors.

Parameters

- **qid** (*str*) – The id of the instance.
- **text** (*str*) – The raw text will be processed with SpaCy.
- **vid** (*int, optional*) – The version, by default 0. When an instance/a target is rewritten, the version will automatically grow.
- **annotator** (*SpacyAnnotator, optional*) – The annotator, by default None. If None, use the default annotator.
- **metas** (*Dict[str, any], optional*) – Additional metas associated with a target, in the format of {key: value}, by default {}

from_bytes() → errudite.targets.target.Target

Change the byte version of the target to normal version. Used for reloading the dump.

Returns The normal version of the target.

Return type *Instance*

generate_id() → str

Get the string key: "qid:{self.qid}-vid:{self.vid}"

Returns The stringed key.

Return type str

get_meta(*meta_name: str*) → Any
Get the meta of the target.

Parameters **meta_name** (*str*) – The name of the meta. Has to be a key in *self.metas*

Returns The meta.

Return type Any

get_text() → str
Get the text associated with the target.

Returns The string

Return type str

key()
Return the key of the target's instance, as a Named Tuple.

Returns The key: `InstanceKey(qid=self.qid, vid=self.vid)`

Return type InstanceKey

serialize() → Dict[str, Any]
Seralize the instance into a json format, for sending over to the frontend.

Returns The serialized version.

Return type Dict[str, Any]

to_bytes() → errudite.targets.target.Target
Change some entries in the target to bytes, for better cache dumping.

Returns The byte version of the target.

Return type *Target*

2.1.2 errudite.targets.label

`Label` is a special subclass of Target, denoting *groundtruth* and *prediction*.

errudite.targets.label.Label

class errudite.targets.label.**Label**(*model: str*, *qid: str*, *text: str*, *vid: int = 0*, *metas: Dict[str, any] = {}*, *annotator: errudite.processor.spacy_annotator.SpacyAnnotator = None*)
`Label` is a special subclass of Target, denoting *groundtruth* and *prediction*. It takes an additional input `model`, which predictor the label is producied by. For groundtruths, make sure this model is `groundtruth`.

Parameters

- **model** (*str*) – Denote the predictor the label is producied by. For groundtruths, make sure this model is `groundtruth`.
- **qid** (*str*) – The id of the instance.
- **text** (*str*) – The raw text will be processed with SpaCy.
- **vid** (*int, optional*) – The version, by default 0. When an instance/a target is rewritten, the version will automatically grow.

- **annotator** (*SpacyAnnotator, optional*) – The annotator, by default None. If None, use the default annotator.
- **metas** (*Dict[str, any], optional*) – Additional metas associated with a target, in the format of {key: value}, by default {}

compute_perform(*groundtruths: Union[Label, List[Label]] = None*, *groundtruths_text: Union[str, List[str]] = None*) → None

Compute the performances of this Label. This function calls `Label.task_evaluator`, and save all the keys and values returned to `self.perform`. **!!** This can be computed with either just `groundtruths` or `groundtruths_text`, but you cannot have neither!

Parameters

- **groundtruths** (*Union[Label, List[Label]], optional*) – The groundtruth Label objects, by default None
- **groundtruths_text** (*Union[str, List[str]], optional*) – The groundtruth string(s), by default None

Returns

Return type None

generate_id() → str

Get the string key: 'qid:{self.qid}-vid:{self.vid}-model:{self.model}'

Returns The stringed key.

Return type str

get_label() → str

Get the label string.

Returns The string

Return type str

get_perform(*perform_name: str = None*) → float

Get a performance metric from this label with the performance name.

Parameters **perform_name** (*str, optional*) – The selected model, by default `None`. If `None`, resolve to `Label.primary_task_metric`.

Returns The performance score. If the perform name does not exist, return 0.

Return type float

is_incorrect() → bool

Check if the prediction is correct. This is done by checking whether the primary metric `Label.task_primary_metric < 1`.

Returns If the model is incorrect.

Return type bool

key() → errudite.targets.interfaces.LabelKey

Return the key of the label as a Named Tuple.

Returns The key: `LabelKey(qid=self.qid, vid=self.vid, model=self.model, label=self.label)`

Return type InstanceKey

classmethod resolve_default_perform_name(*perform_name: str = None*) → str
Resolve the actual selected perform_name. If `perform_name` is given, return perform_name. Otherwise (`perform_name=None or perform_name="DEFAULT"`), return `Label.perform_name`.

Parameters **perform_name** (*str, optional*) – The input named model, by default None

Returns The resolved model

Return type str

set_perform(***kwargs*) → None
All kwargs should be a {name: score} format. Can freely input.

classmethod set_task_evaluator(*task_evaluation_func: Callable[[str, Union[str, List[str]]], Dict[str, float]], task_primary_metric: str*) → None
Because different task has different evaluation metrics and methods, This function sets the evaluation function and primary metric.

Parameters

- **task_evaluation_func** (*Callable[[pred: str, labels: Union[str, List[str]]], Dict[str, float]]*) – the evaluation function that accepts pred and groundtruths, and return a dict of metrics: { metric_name: metric_score }. This is saved as `Label.task_evaluation_func`.
- **task_primary_metric** (*str*) – The primary task metric name, ideally a key of `task_evaluation_func` 's return. This is saved as `Label.task_primary_metric`. This is what DSL resolve to when we set `perform_name="DEFAULT"`

Returns

Return type None

task_evaluation_func(*labels*)
return a dict of metrics: { metric_name: metric_score }

classmethod task_evaluator(*pred: str, labels: Union[str, List[str]]*) → Dict[str, float]
The wrapper for computing the performances.

Parameters

- **pred** (*str*) – The predicted string.
- **labels** (*Union[list, str]*) – The groundtruth or a list of groundtruths.

Returns A dict of metrics: { metric_name: metric_score }

Return type Dict[str, float]

task_primary_metric = 'accuracy'
The primary task metric name, ideally a key of `task_evaluation_func` 's return. This is what DSL resolve to when we set `perform_name="DEFAULT"`

Because *Label* can be of different types (`int`, predefined class `str`, or span `str` extracted from certain targets), we define two subclasses of `Label`.

- `SpanLabel`: To handle tasks like QA, where the output label is a sequence span extracted from input (context), and therefore is not a predefined set. These labels are similarly processed by SpaCy to be queryable.
- `PredefinedLabel`: To handle tasks where the output label are discrete, predefined class types. These outputs will not need any preprocessing.

Subclasses of Label

errudite.targets.label.PredefinedLabel

class errudite.targets.label.**PredefinedLabel**(*model: str, qid: str, text: Union[str, int, float], vid: int = 0, metas: Dict[str, any] = {}, annotator: errudite.processor.spacy_annotator.SpacyAnnotator = None*)

When the label is from a predefined list of numbers or strs, no need to process.

get_text()

Get the label string. This works exactly the same as `self.get_label()`

Returns The string

Return type str

errudite.targets.label.SpanLabel

class errudite.targets.label.**SpanLabel**(*model: str, qid: str, text: str, vid: int = 0, metas: Dict[str, any] = {}, annotator: errudite.processor.spacy_annotator.SpacyAnnotator = None*)

When the label is from a predefined list of numbers or strs, not need to process.

2.1.3 errudite.targets.instance

class errudite.targets.instance.**Instance**(*qid: str, vid: int, rid: str = 'unrewritten', additional_keys: Dict[str, Union[str, int]] = {}*)

`Instances` could be treated as a *wrapper* class of targets, which is used by the DSL to create specific instances. We create instance classes by setting the correct entries and keys created by the targets.

While other entires can flow, **make sure** you set [`predictions` or `prediction`] and [`groundtruths` or `groundtruth`], depending on how many groundtruths you have, and how many models you are using to predict this one instance.

Parameters

- **qid** (*str*) – The id of the instance.
- **vid** (*int, optional*) – The version, by default 0. When an instance/a target is rewritten, the version will automatically grow.
- **rid** (*str, optional*) – The rewrite rule id. If not rewritten (i.e., the original version), it is UNREWRITTEN_RID.
- **additional_keys** (*Dict[str, Union[str, int]], optional*) – Additional keys that can help locate an instance, in the format of {key_name: key}, by default {}

classmethod build_instance_hashes(*instances: List[Instance]*) → Tuple[Dict[errudite.targets.interfaces.InstanceKey, errudite.targets.instance.Instance], Dict[errudite.targets.interfaces.InstanceKey, errudite.targets.instance.Instance], Dict[str, List[errudite.targets.interfaces.InstanceKey]]]

Build the hases

Parameters **instances** (*List[Instance]*) – [description]

Returns [description]

Return type Tuple[Dict[InstanceKey, *Instance*], Dict[InstanceKey, *Instance*], Dict[str, List[InstanceKey]]]

classmethod exists(*key: errudite.targets.interfaces.InstanceKey*, *instance_hash: Dict[errudite.targets.interfaces.InstanceKey, Instance] = {}*, *instance_hash_rewritten: Dict[errudite.targets.interfaces.InstanceKey, Instance] = {}*) → bool

Check whether an instance exists by querying its key.

Parameters

- **key** (*InstanceKey*) – The key of the intended instance.
- **instance_hash** (*Dict[InstanceKey, Instance]*) – A dict that saves all the *original* instances, by default None. It denotes by the corresponding instance keys. If `None`, resolve to `Instance.instance_hash`.
- **instance_hash_rewritten** (*Dict[InstanceKey, Instance]*) – A dict that saves all the *rewritten* instances, by default None. It denotes by the corresponding instance keys. If `None`, resolve to `Instance.instance_hash_rewritten`.

Returns If the instance exists.

Return type bool

from_bytes() → errudite.targets.instance.Instance

Change the byte version of the instance to normal version. Used for reloading the dump.

Returns The normal version of the instance.

Return type *Instance*

generate_id() → str

Get the string key: "qid:{self.qid}-vid:{self.vid}"

Returns The stringed key.

Return type str

classmethod get(*key: errudite.targets.interfaces.InstanceKey*, *instance_hash: Dict[errudite.targets.interfaces.InstanceKey, Instance] = {}*, *instance_hash_rewritten: Dict[errudite.targets.interfaces.InstanceKey, Instance] = {}*) → errudite.targets.instance.Instance

Get the instance by querying its key.

Parameters

- **key** (*InstanceKey*) – The key of the intended instance.
- **instance_hash** (*Dict[InstanceKey, Instance]*) – A dict that saves all the *original* instances, by default None. It denotes by the corresponding instance keys. If `None`, resolve to `Instance.instance_hash`.
- **instance_hash_rewritten** (*Dict[InstanceKey, Instance]*) – A dict that saves all the *rewritten* instances, by default None. It denotes by the corresponding instance keys. If `None`, resolve to `Instance.instance_hash_rewritten`.

Returns The queried instance.

Return type *Instance*

get_all_keys() → Dict[str, Union[str, int]]
Get all the instance keys, including the qid, vid, rid, and all the keys in the additional_keys.

Returns {key_name: key}

Return type Dict[str, Union[int, str]]

get_entry(*entry: str*, *model: str = None*) → errudite.targets.target.Target
Get a target entry from this instance with the entry name.

Parameters

- **entry** (*str*) – The name of the target entry.
- **model** (*str, optional*) – If the entry is "prediction", an additional model string argument can be used to get the specific prediction from a targeted model. By default None

Returns The queried the Target. If non-existing, return `None`.

Return type *Target*

get_perform(*model: str = None*, *perform_name: str = None*) → float
Get the metric of a given model, based on a performance metric name.

Parameters

- **model** (*str, optional*) – The selected model, by default `None`. If `None`, resolve to `Instance.model`.
- **perform_name** (*[type], optional*) – The queried metric name, by default Label.task_primary_metric

Returns The queried metric. If cannot find the prediction from the model, return 0.

Return type float

is_incorrect(*model: str = None*) → bool
Check if the model has a correct prediction. This function gets the prediction (`Label`) from the model, and then call `prediction.is_incorrect()`.

Parameters **model** (*str, optional*) – The selected model, by default `None`. If `None`, resolve to `Instance.model`.

Returns If the model is incorrect.

Return type bool

key() → errudite.targets.interfaces.InstanceKey
Return the key of the instance as a Named Tuple.

Returns The key: `InstanceKey(qid=self.qid, vid=self.vid)`

Return type InstanceKey

classmethod remove_saved(*key: errudite.targets.interfaces.InstanceKey*) → bool
Remove the saved instance from the hashes (`Instance.instance_hash`, `Instance.instance_hash_rewritten`, and `Instance.qid_hash`) by querying its key.

Parameters **key** (*InstanceKey*) – The key of the intended instance.

Returns True if correctly removed (or not exist.)

Return type bool

classmethod resolve_default_model(*model: str = None*) → str
Resolve the actual selected model. If `model` is given, return model. Otherwise (`model=None or model="ANCHOR"`), return `Instance.model`.

Parameters **model** (*str, optional*) – The input named model, by default None

Returns The resolved model

Return type str

classmethod resolve_default_rewrite(*rid: str = None*)
Resolve the actual selected rewrite rule. If `rid` is given, return rid. Otherwise (`rid=None or rid="SELECTED"`), return `Instance.selected_rewrite`.

Parameters **rid** (*str, optional*) – The input named rewrite rule string, by default None

Returns The resolved model

Return type str

classmethod save(*instance: errudite.targets.instance.Instance*) → bool
Save an instance into the hash: (`Instance.instance_hash`, `Instance.instance_hash_rewritten`, and `Instance.qid_hash`)

Parameters **instance** (*Instance*) – The instance to be saved.

Returns If the instance is correctly saved.

Return type bool

serialize() → Dict[str, Any]
Seralize the instance into a json format, for sending over to the frontend.

Returns The serialized version.

Return type Dict[str, Any]

classmethod set_default_model(*model: str*) → None
Set a default model for the whole class instance to share.

Parameters **model** (*str*) – The model name.

Returns

Return type None

classmethod set_default_rewrite(*rid: str*)
Set a default rewrite rule for the whole class instance to share.

Parameters **model** (*str*) – The rewrite rule names.

Returns

Return type None

set_entries(***kwargs*) → None
Save the targets as entries of the target, so the instance can serve as the wrapper. It accepts `**kwargs`, so the names of the targets can be easily customized. It's supposed to be called by `instance.set_entries(target_name1=target1, target_name2=target2)`.

Returns [description]

Return type None

classmethod set_entry_keys(*entries: List[str]*) → None
Save the target entry names used for a specific task to `Instance.instance_entries`.

Parameters **entries** (*List[str]*) – A list of entry names.

show_instance() → None
Print an instance string that represent the key information of the instance, including the keys and the entries.

Returns

Return type None

to_bytes() → errudite.targets.instance.Instance
Change some entries in the instance to bytes, for better cache dumping.

Returns The byte version of the instance.

Return type *Instance*

instance_entries = []
`List[str]`, The names of the entry targets saved in the Instance.

instance_hash = {}
`Dict[InstanceKey, Instance]`, A dict that saves all the *original* instances, denoted by the corresponding instance keys.

instance_hash_rewritten = {}
`Dict[InstanceKey, Instance]`, A dict that saves all the *rewritten* instances, denoted by the corresponding instance keys.

ling_perform_dict = {}
The relationship between linguistic features and model performances.

Type `dict`

qid_hash = {}
`Dict[str, List[InstanceKey]]`, A dict that denotes wraps different versions of instance keys

selected_rewrite = 'unrewritten'
`str`, The selected rewrite. This is what DSL resolve to when we set `rewritten="SELECTED"`

train_freq = {}
`Dict[str, int]` The training vocabulary frequency

We provide some preprocessed cache folders downloading:

2.2 Download Precomputed Cache Folders

If you would like to try out Errudite without doing the preprocessing your own, Errudite has several precomputed cache folders that can be easily downloaded and used:

```
python -m errudite.download

Commands:
    cache_folder_name
                A folder name. Currently, we allow downloading the following:
                squad-100, squad-10570.
    cache_path  A local path where you want to save the cache folder to.
```

After the downloading, the data will be in `{cache_path}/{cache_folder_name}/`. The cache folder is in the following structure:

CHAPTER

THREE

MAIN ANALYSIS METHODS

At the core of Errudite is an expressive domain-specific language (DSL) for precisely querying instances based on linguistic features. Using composable building blocks in DSL, Errudite supports forming semantically meaningful groups and rewriting instances to test counterfactuals across all available validation data.

3.1 Domain Specific Language (DSL)

Errudite has a DSL language that:

1. Use primitive functions that run on targets to extract fundamental instance metadata (e.g., `length(premise)` returns the length of a question).
2. Allows string command inputs, which can be automatically parsed into actual primitive functions. The objective here is, we try to query the frequently used targets more easily for you. On a high level, parser works as the following:
   - If it recognizes a target name that occurrs in *instance.entries*, it automatically retrive the target.
   - If it recognizes a registered primitive function, it runs the function.
   - It also supports more general operators like >, <=, `and`, `or`, etc.
   - It resolves previously created attributes (`attr:attr_name`) and groups (`group:group_name`).

There is a basic wrapper class, `errudite.build_blocks.PrimFunc`, to wrap all the functions up. Errudite also has a list of functions to support computing instance attributes and/or build instance groups.

3.1.1 Wrapping class for prim functions

class errudite.build_blocks.prim_func.**PrimFunc**

A wrapper function primitive functions used in the domain specific language. It inherits `errudite.utils.registrable`, so all the functions can be registered to this class with their function names.

classmethod build_instance_func(*func_name: str*, *instance: errudite.targets.instance.Instance*) → Callable

Given an instance, adjust the one function to fit specifically for that instance. It does it by automatically filling in the inputs that have the same variable name as a target entry in the instance, so users could write the DSL function more easily.

Parameters

- **func_name** (*str*) – The name of the function.
- **instance** (*Instance*) – The instance that all the function should be adjusted to.

Returns The functions with entries filled in.

Return type Callable

classmethod build_instance_func_list(*instance: errudite.targets.instance.Instance*) → Dict[str, Callable]

Given an instance, adjust the all the functions to fit specifically for that instance. It does it by automatically filling in the inputs that have the same variable name as a target entry in the instance, so users could write the DSL function more easily.

instance [Instance]

The instance that all the function should be adjusted to.

Dict[str, Callable] A dict of functions, with each function being partially filled in (i.e. users only need to fill in variables whose name do not occur in the entries of instances.)

classmethod get_funcs_hash() → Dict[str, List[str]]

This is an inspection function. By calling this, we construct a dict that presents `{ func_name : [ args ] }`, to show what functions are available, and what arguments and keyword arguments needed.

Returns The inspection dict hash.

Return type Dict[str, List[str]]

3.1.2 Pre-implemented prim functions

Errudite has a list of functions to support computing instance attributes and/or build instance groups. These functions are called `primitive functions` — they are *attribute extractors* act on *targets* to extract fundamental instance metadata (e.g., *length(q)* returns the length of a question). These include:

- basic extractors like length,
- general purpose linguistic features like token `LEMMA`, `POS` tags, and entity (`ENT_TYPE`) annotations,
- standard prediction performance metrics such as `f1` or `accuracy`,
- between-target relations such as `overlap(t1, t2)`, and
- domain-specific attributes (e.g., for Machine Comprehension or VQA) such as `question_type` and `answer_type`.

`prim_funcs` are composable through standard logical and numerical operators, serving as building blocks for more complex attributes.

Converters and targets

Get targets: These targets contain text spans post-processed with state-of-the-art POS taggers, lemmatizers and NER models, along with metadata such as example id, or (in the answer case) the model that generated it. When additional metadata is not used, `Target` can be treated just as `Span` in a function, or a piece of text with its linguistic features.

question|context|groundtruth → Target

Automatically query the target object (`Question` and `Answer` in Visual Question Answering and Machine Comprehension, as well as `Context` in Machine comprehension). This can be easily extended to any key that is in `Instance.instance_entries`.

errudite.build_blocks.prim_funcs.get_prediction.**prediction**(*model: str, predictions: Union[Label, List[Label]]*) → Label

Get the prediction object of a given model.

Parameters

- **model** (*str*) – The model to query.
- **predictions** (*Union[Label, List[Label]]*) – All the predictions available. *Automatically filled in when using the DSL parser.*

Returns The selected prediction.

Return type *Label*

Converters that extract sub-spans, short phrases, or sentences from targets.

errudite.build_blocks.prim_funcs.token.**token**(*docs: Union[spacy.tokens.Span, Target], idxes: Union[int, List[int]] = None, pattern: Union[str, List[str]] = None*) → Union[spacy.tokens.Span, spacy.tokens.Token]

Get a list of tokens from the target based on idxes (sub-list) and pattern. Note that `idxes` runs before `pattern`. That is, if the idxes exist, the pattern filters the idxed doc tokens.

Parameters

- **docs** (*Union[Target, Span]*) – The doc to be queried.
- **idxes** (*Union[int, List[int]], optional*) – Retrive the sub-list of tokens from docs, with idx(es). By default None
- **pattern** (*Union[str, List[str]], optional*) – Used to filter and get the sub-list of spans in the doc span list. Pattern allows linguistic annotations and automatically detects queries on POS tags and entity types, in ALL CAPS. For example, `(what, which) NOUN)` may query all docs that have `what NOUN` or `which NOUN`. If a list, then all the patterns in a list are "OR". By default None

Returns The queried sub-list.

Return type Union[Span, Token]

errudite.build_blocks.prim_funcs.get_sentence.**sentence**(*answer: QAAnswer, context: Context, shift: Union[int, List[int]] = 0*) → spacy.tokens.Span

Machine Comprehension only Get the sentence that contains a given answer. Shift indicates if neighboring sentences should be included.

Parameters

- **answer** (*QAAnswer*) – The selected answer.
- **context** (*Context*) – The context target of a given instance. *Automatically filled in when using the DSL parser.*
- **shift** (*Union[int, List[int]], optional*) – Shift indicates if neighboring sentences should be included, by default 0 If `shift==0`, then the actual sentence is returned; if `shift==[-2, -1,1,2]`, then the four sentences surrounding the answer sentence are returned.

Returns The selected sentence that contains the answer.

Return type Span

General computation

errudite.build_blocks.prim_funcs.digits.**abs_num**(*number: Union[int, float]*) → Union[int, float]

Returns the absolute value.

Parameters **number** (*Union[int, float]*) – The input number.

Returns The output, absoluted number.

Return type Union[int, float]

errudite.build_blocks.prim_funcs.digits.**digitize**(*target: Union[str, int, float]*) → Union[int, float]

Parses an input into a number if `is_digit(input) == True`; Otherwise returns `None`.

Parameters **target** (*Union[str, int, float]*) – the input to be digitized.

Returns The digitized version of target. If not a number, return `None`.

Return type Union[int, float]

errudite.build_blocks.prim_funcs.digits.**is_digit**(*target: Union[str, int, float]*) → bool

Determines if an input is a number, or – in the case of a string input – if it can be parsed into a number.

Parameters **target** (*Union[str, int, float]*) – The input to check if is a digit.

Returns Whether or not it's a digit.

Return type bool

errudite.build_blocks.prim_funcs.digits.**truncate**(*value: Union[int, float], min_value: Union[int, float] = -1, max_value: Union[int, float] = 50*) → int

Clamps a given number to a given domain.

Parameters

- **value** (*Union[int, float]*) – The value to be clamped.
- **min_value** (*Union[int, float], optional*) – The minumum number allowed, by default -1
- **max_value** (*Union[int, float], optional*) – The maximum number allowed, by default 50

Returns The clamped value.

Return type int

errudite.build_blocks.prim_funcs.length.**length**(*docs: Union[Target, spacy.tokens.Span, List[Union[Target, spacy.tokens.Span]]]*) → int

The length of a given span, in tokens. If the input is a List, take the min length of all spans in the list.

Parameters **docs** (*Union[Target, Span, List[Union[Target, Span]]]*) – The input doc(s) for computing the length.

Returns The length.

Return type int

errudite.build_blocks.prim_funcs.freq.**freq**(*target: Union[Target, spacy.tokens.Span], target_type: str*) → float

Returns the frequency of a token occurring in the training data, given a target type

Parameters

- **target** (*Union[Target, Span]*) – The targeted token.

- **target_type** (*str, optional*) – Needs to be a key in `Instance.train_freq` to help determine the frequency dictionary.

Returns [description]

Return type float

`errudite.build_blocks.prim_funcs.get_meta.`**`get_meta`**(*target: errudite.targets.target.Target*, *meta_name: str*) → Any

Query the extra meta in a target.

Parameters

- **target** (*Target*) – A given target object.
- **meta_name** (*str*) – The name of the metadata. Has to be a key in `target.metas`.

Returns The queried meta.

Return type Any

`errudite.build_blocks.prim_funcs.similar_token.`**`find_similar_token`**(*word: Union[spacy.tokens.Token, str]*, *search_type: str*) → str

Find related words from wordnet, given a token's text and POS. If cannot find one, return itself.

When using the DSL parser, this function can be called in alternative ways, with `search_type` being automatically filled in: `[get_synonym|get_antonym](word)`.

Parameters

- **word** (*Union[Token, str]*) – The given word. Can be a spacy token or just a string (in which case, the POS tag will not be specified.)
- **search_type** (*str*) – "synonym" or "antonym".

Returns The synonym or the antonym string.

Return type str

`errudite.build_blocks.prim_funcs.apply.`**`apply`**(*func: Callable*, *rewrite: str = 'SELECTED'*) → Any

Applies query functions to instances rewritten by the named rule rewrite.

Parameters

- **func** (*Callable*) – A query function.
- **rewrite** (*str, optional*) – Use a named rule rewrite to get instances rewritten by the rule. If using "SELECTED", it will be automatically resolved to `Instance.model` , by default "SELECTED"

Returns return the corresponding output format as the `func`.

Return type Any

`errudite.build_blocks.prim_funcs.is_rewritten_by.`**`is_rewritten_by`**(*instance: Instance*, *rewrite: str = 'SELECTED'*) → bool

Test if an instance is generated by the named rewrite rule.

Parameters

- **instance** (*Instance*) – The instance to be tested. *Automatically filled in when using the DSL parser.*
- **rewrite** (*str, optional*) – Use a named rule rewrite to get instances rewritten by the rule. If using "SELECTED", it will be automatically resolved to `Instance.model` , by default "SELECTED"

Returns If an instance is generated by the named rewrite rule.

Return type bool

Linguistic attributes

`errudite.build_blocks.prim_funcs.linguistic.`**`LABEL`**(*target: Label*) → str

Get the raw string from a label target.

Parameters **target** (*Label*) – The label object (target) to be converted to string.

Returns The string.

Return type str

`errudite.build_blocks.prim_funcs.linguistic.`**`STRING`**(*target: Union[Target, spacy.tokens.Span]*) → str

Get the raw string from a given span or target.

Parameters **target** (*Union[Target, Span]*) – The target to be converted to string.

Returns The string.

Return type str

`errudite.build_blocks.prim_funcs.linguistic.`**`linguistic`**(*spans: Union[Target, spacy.tokens.Span], label: str = 'lemma', pattern: Union[str, List[str]] = None, get_root: bool = False, get_most_common: bool = False*) → Union[str, List[str]]

Return the specified linguistic feature of a span with one more more tokens.

If `pattern` is provided, it is used to filter and get the sub-list of spans in the span list. For example, if `pattern=="NOUN"`, then the overlap will only be on tokens with a `NOUN` tag.

If `get_root==True`, gets the single linguistic feature of the *primary* token, or the one within the ground truth span that is highest in the dependency parsing tree.

When using the DSL parser, this function can be called in the following alternative ways, with `label` being automatically filled in: `[LEMMA|POS|TAG|ENT](spans, pattern, get_root)`.

Parameters

- **spans** (*Union[Target, Span]*) – The span or target to get the info.
- **label** (*str, optional*) – The linguistic feature. Could be `lemma, ent_type, pos`.
- **pattern** (*Union[str, List[str]], optional*) – Query the specific pattern, by default None
- **get_root** (*bool, optional*) – If to get the single linguistic feature of the *primary* token, by default False

- **get_most_common** (*bool, optional*) – If to get the most frequently occurred linguistic feature, by default False

Returns The linguistic feature, or feature list if (1) multiple spans are given, and (2) *get_root* and *get_most_common* are both false.

Return type Union[str, List[str]]

Performance Metrics

errudite.build_blocks.prim_funcs.perform.**perform**(*model: str, predictions: Union[Label, List[Label]], perform_name: str*) → float

Get the specified performance metric for one instance, given the selected model. *When using the DSL parser*, this function can be called in alternative ways, with `perform_name` being automatically filled in:

- `[f1|exact match|precision|recall|accuracy|confidence]`, with get being the corresponding metrics. Confidence is for usually the model prediction probability.
- `is_correct_sent`, with `get=sent`.

Parameters

- **model** (*str*) – The model to query.
- **predictions** (*Union[Label, List[Label]]*) – All the predictions available. *Automatically filled in when using the DSL parser.*
- **perform_name** (*str*) – The selected metric name. It has to be a key that's in `label.perform`.

Returns The queried metric.

Return type float

Between-target relations

errudite.build_blocks.prim_funcs.overlap.**overlap**(*doc_a: Union[Target, spacy.tokens.Span], doc_b: Union[Target, spacy.tokens.Span], label: str = 'lemma', return_token_list: bool = False*) → Union[float, List[spacy.tokens.Token]]

A directional overlapping: returns the ratio of tokens in `doc_a` that also occur in `doc_b` (`len(doc_a & doc_b) / len(doc_a)`)

Parameters

- **doc_a** (*Union[Target, Span]*) – One target/span in the computation.
- **doc_b** (*Union[Target, Span]*) – One target/span in the computation.
- **label** (*str, optional*) – Determines what linguistic feature both docs to be converted to, to do the overlap computation, by default 'lemma'
- **return_token_list** (*bool, optional*) – To return the actual token list if True, or the ratio if faulse, by default False

Returns Either the ratio, or the actual overlapping token list.

Return type Union[float, List[Token]]

Domain-specific attributes

errudite.build_blocks.prim_funcs.dep_distance.**dep_distance**(*target: Union[Answer, List[Answer]], question: Question, context: Context, pattern: Union[str, List[str]] = None*) → float

Machine Comprehension only Dependency distance between a key question token and the answer token. The key is computed by finding tokens that do not occur frequently in the context and is not far from the given answer.

Parameters

- **target** (*Union[Answer, List[Answer]]*) – A selected answer object (Or a list.)
- **question** (*Question*) – The question target of a given instance. *Automatically filled in when using the DSL parser.*
- **context** (*Context*) – The context target of a given instance. *Automatically filled in when using the DSL parser.*
- **pattern** (*Union[str, List[str]], optional*) – Fixes the keyword linguistic feature., by default None

Returns The distance.

Return type float

errudite.build_blocks.prim_funcs.logic_operations.**count**(*vars: List[Any]*) → int

Count the number of members in the input list.

Parameters **vars** (*List[Any]*) – The vars to be counted.

Returns The counted number.

Return type int

errudite.build_blocks.prim_funcs.logic_operations.**has_all**(*container: List[Any], contained: List[Any]*) → bool

Determines whether one list container contains all of the members present in another lists.

Parameters

- **container** (*List[Any]*) – The container list, or the super set.
- **contained** (*List[Any]*) – The contained list, or the subset.

Returns If the 'all' condition holds.

Return type bool

errudite.build_blocks.prim_funcs.logic_operations.**has_any**(*container: List[Any], contained: List[Any]*) → bool

Determines whether one list container contains any of the members present in another lists.

Parameters

- **container** (*List[Any]*) – The container list, or the super set.
- **contained** (*List[Any]*) – The contained list, or the subset.

Returns If the 'any' condition holds.

Return type bool

errudite.build_blocks.prim_funcs.offset.**answer_offset**(*pred: QAAnswer, groundtruths: Union[QAAnswer, List[QAAnswer]], context: Context, direction: str = 'left', get: str = 'delta'*) → Union[spacy.tokens.Span, int]

Machine Comprehension only Compute the offset between prediction and ground truth in the left or right direction. Depending on `get`, this function returns either the offset spans, or the position differences.

When using the DSL parser, this function can be called in alternative ways, with `get` being automatically filled in: `[answer_offset_delta|answer_offset_span](...)`.

Parameters

- **pred** (*QAAnswer*) – The selected prediction.
- **groundtruths** (*Union[QAAnswer, List[QAAnswer]]*) – The groundtruth(s). *Automatically filled in when using the DSL parser.*
- **context** (*Context*) – The context object where the `pred` and `groundtruths` come from. *Automatically filled in when using the DSL parser.*
- **direction** (*str, optional*) – Compute the delta between the start idx of spans if 'left', or the end idx of spans if 'right', by default 'left'.
- **get** (*str, optional*) – Determines the output type. If 'delta', return the position differences (`int`). If 'span', return the actual spans, by default 'delta'

Returns Either the differing spans or the position difference.

Return type Union[Span, int]

errudite.build_blocks.prim_funcs.types.**answer_type**(*target: Answer*) → str

Machine Comprehension only Returns the answer type, computed based on TREC (Li and Roth, 2002) and the named entities of the answer. Returns one of the following: ABBR, DESC, ENTY, HUM, LOC, NUM.

Parameters **target** (*Answer*) – An answer target

Returns The answer type.

Return type str

errudite.build_blocks.prim_funcs.types.**question_type**(*target: Question*) → str

Machine Comprehension only Returns the question type: either the WH-word or the first word in a sentence.

Parameters **target** (*Question*) – A question target.

Returns The question type.

Return type str

3.2 Attributes and Groups

Errudite's DSL and operators can be used to:

1. extract complex attributes from instances, and
2. create semantically meaningful groups.

3.2.1 errudite.builts.built_block

3.2.2 errudite.builts.attribute

3.2.3 errudite.builts.group

3.3 Rewrite Rules

For scalable counterfactual analysis, Errudite uses rules to rewrite all relevant instances within a group – similar to search and replace but with the flexibility and power of the Errudite DSL.

All the rules are defined under `errudite.rewrites.rewrite.Rewrite`. This is a subclass of *errudite.utils.registrable.Registrable* and all the actual rewrite rule classes are registered under `Rewrite` by their names.

3.3.1 Base class

There are three large types of rewrite rules: (1) defaults, (2) those implemented in the syntax of `rewrite(target, from -> to)`, and (3) those that allow customized raw python functions.

3.3.2 Rewrite rules defined by DSL

A rewrite rule is specified using the syntax `rewrite(target,from -> to)`, where target indicates the part of the instance that should be rewritten by replacing from with to. Depending on whether or not you want to use linguistic features, you could use either `RewriteStr`, or `RewritePattern`.

errudite.rewrites.replace_str

errudite.rewrites.replace_pattern

3.3.3 Rewrite with customized functions

3.3.4 Default rewrite rules

For convenience, Errudite also includes default rules.

errudite.rewrites.remove_clue

errudite.rewrites.remove_context_sent

errudite.rewrites.resolve_coref

CHAPTER

FOUR

EXTENDING ERRUDITE

To extend Errudite to your own task and model, you will need to write your own `DatasetReader`, and your own `Predictor` wrapper. A `DatasetReader` knows how to turn a file containing a dataset into a collection of `Instance` s, and how to handle writting the processed instance caches to the cache folders. A `Predictor` wraps the prediction function of a model, and transfers the prediction to `Label` targets.

4.1 Extensible Dataset Readers

To extend Errudite to different tasks, We implement the dataset reader in a way such that it can be extended to customized dataset handlers.

All the predictors are defined under `errudite.io.dataset_reader.DatasetReader`. This is a subclass of `errudite.utils.registrable.Registrable` and all the actual reader classes are registered under `DatasetReader` by their names.

4.1.1 Dataset Reader Base Class

class errudite.io.dataset_reader.**DatasetReader**(*cache_folder_path: str = None*)

Adjusted from https://allenai.github.io/allennlp-docs/api/allennlp.data.dataset_readers.html A `DatasetReader` knows how to turn a file containing a dataset into a collection of `Instance` s.

It also handles writting the processed instance caches to the cache folders. When fully processed, the cache folder contains several folders/files:

```
.
├── analysis # saved attr, group and rewrite json that can be reloaded.
│   ├── save_attr.json
│   ├── save_group.json
│   └── save_rewrite.json
├── evaluations # predictions saved by the different models, with the model name␣
↪being the folder name.
│   └── bidaf.pkl
├── instances.pkl # Save all the `Instance`, with the processed Target.
│   # A dict saving the relationship between linguistic features and model␣
↪performances.
│   # It's used for the programming by demonstration.
├── ling_perform_dict.pkl
├── train_freq.json # The training vocabulary frequency
└── vocab.pkl # The SpaCy vocab information.
```

To implement your own, just override the *self._read* method to return a list of the instances. This is a subclass of *errudite.utils.registrable.Registrable* and all the actual rewrite rule classes are registered under `Rewrite` by their names.

Parameters **cache_folder_path** (*str, optional*) – Set the cache folder path, by default None. If not given, the default is `./caches/`.

compute_ling_perform_dict (*instances: List[errudite.targets.instance.Instance]*) → None

Compute the relationship between linguistic features and model performances. It's used for the programming by demonstration.

Parameters **instances** (*List[Instance]*) – A list of instances.

Returns

The result is saved to `Instance.ling_perform_dict`. It's in the format of:

```
{
    target_name: {
        pattern: {
            model_name: {
                cover: how many instances are there.
                err_cover: The ratio of incorrect predictions with␣
↪the pattern, overall all the incorrect predictions.
                err_rate: the ratio of incorrect predictions, over␣
↪all the instances wit the pattern.
            }
        }
    }
}
```

Return type None

count_vocab_freq (*file_path: str*) → None

Compute the vocabulary from a given data file. This is for getting the training frequency and save to `Instance.train_freq`. This function calls `self._read` with `lazy=False`.

Parameters **file_path** (*str*) – The path of the input data file. We suggest using the training file!

Returns

Return type None

create_instance (*qid: str, additional_keys: Dict[str, Union[str, int]] = {}, **targets*) → errudite.targets.instance.Instance

Create an instance given the qid, additional keys, and targets and kwargs.

Parameters

- **qid** (*str*) – The id of the instance.
- **additional_keys** (*Dict[str, Union[str, int]], optional*) – Additional keys that can help locate an instance, in the format of {key_name: key}, by default {}
- **targets** – A list of targets given in the format of target_name=target: Target.

Returns An instance

Return type *Instance*

dump_preprocessed () → None

Save all the preprocessed information to the cache file. It includes `instances.pkl`, `ling_perform_dict.pkl`, `vocab.pkl`, and all the `evaluations/[predictor_name].pkl`.

Returns [description]

Return type None

load_preprocessed(*selected_predictors: List[str] = None*) → None
Re-store all the preprocessed information. In specific, it reloads:

- Instances
- Set the predictions from models as entries of the instances, and set `Instance.instance_hash`, `Instance.instance_hash_rewritten`, and `Instance.qid_hash`.
- Get the `Instance.ling_perform_dict`, which saves the relationship between linguistic features and model performances, and `Instance.train_freq`, which saves the training vocabulary frequency.

Parameters **selected_predictors** (*List[str], optional*) – If set, only load the predictions from the selected predictors. Otherwise, load all the predictors in *cache_path/evaluations*. By default None

Returns

Return type None

read(*file_path: str, lazy: bool = False, sample_size: int = None*) → List[errudite.targets.instance.Instance]
Returns a list containing all the instances in the specified dataset.

Parameters

- **file_path** (*str*) – The path of the input data file.
- **lazy** (*bool, optional*) – If `lazy==True`, only run the tokenization, does not compute the linguistic features like POS, NER. By default False
- **sample_size** (*int, optional*) – If sample size is set, only load this many of instances, by default None

Returns The instance list.

Return type List[*Instance*]

We have several defaul implementations for several different tasks, and for some tasks, we also have some default, supporting predictor impelmentations (especially those from `Allennlp`.)

4.1.2 Task-Specific Readers

SQAuD v1.1 Reader

class errudite.io.squad_reader.**SQUADReader**(*cache_folder_path: str = None*)
This loads the data from squad v1.1, for the Machine Comprehension task: https://rajpurkar.github.io/SQuAD-explorer/.

- default evaluation metric: f1
- target entries in an instance: question, context, predictions, groundtruths.

This can be queried via:

```
from errudite.io import DatasetReader
DatasetReader.by_name("squad")
```

SST Reader

class errudite.io.sst_reader.**SSTReader**(*cache_folder_path: str = None*, *use_subtrees: bool = False*, *granularity: str = '5-class'*)

This loads the data from The Stanford Sentiment Treebank (SNLI) Corpus: https://nlp.stanford.edu/sentiment/treebank.html

- default evaluation metric: accuracy
- target entries in an instance: query, predictions, groundtruth.

This can be queried via:

```
from errudite.io import DatasetReader
DatasetReader.by_name("sst")
```

SNLI Reader

class errudite.io.snli_reader.**SNLIReader**(*cache_folder_path: str = None*)

This loads the data from The Stanford Natural Language Inference (SNLI) Corpus: https://nlp.stanford.edu/projects/snli/

- default evaluation metric: accuracy
- target entries in an instance: hypothesis, premise, predictions, groundtruth.

This can be queried via:

```
from errudite.io import DatasetReader
DatasetReader.by_name("snli")
```

4.2 Extensible Predictors

Many analyses in Errudite rely on real-time models predictions (especially the rewritings). We implement predictors in a way such that it can be extended to customized predictors.

All the predictors are defined under `errudite.predictors.predictor.Predictor`. This is a subclass of `errudite.utils.registrable.Registrable` and all the actual predictor classes are registered under `Predictor` by their names.

We also have an Allennlp predictor wrapper.

4.2.1 Predictor Base Class

class errudite.predictors.predictor.**Predictor**(*name: str*, *description: str*, *model: any*, *perform_metrics: List[str]*)

A base class for predictors. A predictor runs prediction on raw texts and also instances. It also saves the performance score for the predictor.

This is a subclass of `errudite.utils.registrable.Registrable` and all the actual rewrite rule classes are registered under `Predictor` by their names.

Parameters

- **name** (*str*) – The name of the predictor.
- **description** (*str*) – A sentence describing the predictor.

- **model** (*any*) – The executable model.
- **perform_metrics** (*List[str]*) – The name of performance metrics.

perform

```
{ perform_name: the averaged performance score. }
```

Type Dict[str, float]

classmethod create_from_json (*raw: Dict[str, str]*) → errudite.predictors.predictor.Predictor
Recreate the predictor from its seralized raw json.

Parameters **raw** (*Dict[str, str]*) – The json version definition of the predictor, with name, description, model_path, and model_online_path.

Returns The re-created predictor.

Return type *Predictor*

evaluate_performance (*instances: List[Instance]*) → None
Save the performance of the predictor. It iterates through metric names in `self.perform_metrics`, and average the corresponding metrics in `instance.prediction.perform`. It saves the results in `self.perform`.

Parameters **instances** (*List[Instance]*) – The list of instances, with predictions from this model already saved as part of its entries.

Returns The result is saved in `self.perform`.

Return type None

classmethod model_predict (*predictor: Predictor*, ***targets*) → Label
Define a class method that takes Target inputs, run model predictions, and wrap the output prediction into Labels.

Parameters

- **predictor** (*Predictor*) – A predictor object, with the predict method implemented.
- **targets** (*Target*) – Targets in kwargs format

Returns The predicted output, with performance saved.

Return type *Label*

Raises **NotImplementedError** – This needs to be implemented per task.

predict (***kwargs*)
run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

serialize () → Dict
Seralize the instance into a json format, for sending over to the frontend.

Returns The serialized version.

Return type Dict[str, Any]

We have several defaul implementations for several different tasks, and for some tasks, we also have some default, supporting predictor impelmentations (especially those from Allennlp.)

4.2.2 Question Answering/Machine Comprehension

Task-oriented base class

class errudite.predictors.qa.predictor_qa.**PredictorQA**(*name: str*, *description: str*, *model: any*)

Predictor wrapper for question answering/machine comprehension tasks. Perform metrics: `['f1', 'em', 'sent', 'precision', 'recall', 'confidence']`

This can be queried via:

```
from errudite.predictors import Predictor
Predictor.by_name("qa_task_class")
```

classmethod model_predict(*predictor: Predictor*, *question: Question*, *context: Context*, *groundtruths: List[QAAnswer]*) → QAAnswer

Define a class method that takes Target inputs, run model predictions, and wrap the output prediction into Labels.

Parameters

- **predictor** (*Predictor*) – A predictor object, with the predict method implemented.
- **question** (*Question*) – Question target.
- **context** (*Context*) – Context target.
- **groundtruths** (*List[QAAnswer]*) – A list of groundtruths, typed QAAnswer.

Returns The predicted output, with performance saved.

Return type QAAnswer

predict(*qtext: str*, *ptext: str*) → Dict[str, float]

run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

More specific model implementations

class errudite.predictors.qa.predictor_bidaf.**PredictorBiDAF**(*name: str*, *model_path: str = None*, *model_online_path: str = None*, *description: str = ''*)

The wrapper for BidirectionalAttentionFlow model, as implemented in Allennlp: https://allenai.github.io/allennlp-docs/api/allennlp.predictors.html#bidaf

This can be queried via:

```
from errudite.predictors import Predictor
Predictor.by_name("bidaf")
```

predict(*qtext: str*, *ptext: str*) → Dict[str, float]

run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

4.2.3 Visual Question Answering

Task-oriented base class

class errudite.predictors.vqa.predictor_vqa.**PredictorVQA**(*name: str*, *description: str*, *model: any*)

Predictor wrapper for visual question answering tasks. Perform metrics: `['accuracy', 'confidence']`

classmethod model_predict(*predictor: PredictorVQA*, *question: Question*, *groundtruths: List[VQAAnswer]*) → VQAAnswer

Define a class method that takes Target inputs, run model predictions, and wrap the output prediction into Labels.

Parameters

- **predictor** (*Predictor*) – A predictor object, with the predict method implemented.
- **query** (*Target*) – A sentence, transferred to the target.
- **groundtruth** (*List[VQAAnswer]*) – A list of groundtruths, typed VQAAnswer.

Returns The predicted output, with performance saved.

Return type VQAAnswer

predict(*qtext: str*, *img_id: str*) → Dict[str, float]

run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

More specific model implementations

4.2.4 Natural Language Inference

Task-oriented base class

class errudite.predictors.nli.predictor_nli.**PredictorNLI**(*name: str*, *description: str*, *model: any*)

Predictor wrapper for natural language inference tasks. perform metrics: `['accuracy', 'confidence']`

This can be queried via:

```
from errudite.predictors import Predictor
Predictor.by_name("nli_task_class")
```

classmethod model_predict(*predictor: PredictorNLI*, *premise: Target*, *hypothesis: Target*, *groundtruth: Label*) → Label

Define a class method that takes Target inputs, run model predictions, and wrap the output prediction into Labels.

Parameters

- **predictor** (*Predictor*) – A predictor object, with the predict method implemented.
- **premise** (*Target*) – The premise target.
- **hypothesis** (*Target*) – The hypothesis target.
- **groundtruth** (*Label*) – A groundtruth, typed Label.

Returns The predicted output, with performance saved.

Return type *Label*

predict (*premise: str*, *hypothesis: str*) → Dict[str, float]
run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

More specific model implementations

class errudite.predictors.nli.predictor_decompose_att.**PredictorDecomposeAtt**(*name: str*, *model_path: str = None*, *model_online_path: str = None*, *description: str = ''*)

The wrapper for DecomposableAttention model, as implemented in Allennlp: https://allenai.github.io/allennlp-docs/api/allennlp.predictors.html#decomposable-attention

This can be queried via:

```
from errudite.predictors import Predictor
Predictor.by_name("snli")
```

predict (*premise: str*, *hypothesis: str*) → Dict[str, float]
run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

4.2.5 Sentiment Analysis

Task-oriented base class

class errudite.predictors.sentiment_analysis.predictor_sentiment_analysis.**PredictorSA**(*name: str*, *description: str*, *model: any*)

Predictor wrapper for sentiment analysis tasks. Perform metrics: `['accuracy', 'confidence']`

This can be queried via:

```
from errudite.predictors import Predictor
Predictor.by_name("sentiment_task_class")
```

classmethod model_predict (*predictor: Predictor*, *query: Target*, *groundtruth: Label*) → Label
Define a class method that takes Target inputs, run model predictions, and wrap the output prediction into Labels.

Parameters

- **predictor** (*Predictor*) – A predictor object, with the predict method implemented.
- **query** (*Target*) – A sentence, transferred to the target.
- **groundtruth** (*Label*) – A groundtruth, typed Label.

Returns The predicted output, with performance saved.

Return type *Label*

predict (*premise: str*, *hypothesis: str*) → Dict[str, float]
run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

More specific model implementations

class errudite.predictors.sentiment_analysis.predictor_bcn.**PredictorBCN**(*name: str*, *model_path: str = None*, *model_online_path: str = None*, *description: str = ''*)
The wrapper for a sentiment analysis model, as implemented in Allennlp: https://allenai.github.io/allennlp-docs/api/allennlp.predictors.html#text-classifier

This can be queried via:

```
from errudite.predictors import Predictor
Predictor.by_name("bcn")
```

predict (*query: str*) → Dict[str, float]
run the prediction.

Raises **NotImplementedError** – Should be implemented in subclasses.

Errudite can be used in JupyterLab, or, for machine comprehension and visual question comprehension, we have a graphical user interface:

4.3 Run the Graphical User Interface

The following command runs the visual GUI (note that currently we only support machine comprehension. We will support visual question answering shortly):

```
python -m errudite.server

Commands:
    config_file     A yaml config file path.
```

The config file looks like the following:

```
task: qa # the task, should be "qa" and "vqa".
cache_path: {cache_path}/{cache_folder_name}/ # the cached folder.
model_metas: # a model.
- name: bidaf
  model_class: bidaf # an implemented model class
  model_path: # a local model file path
  # an online path to an Allennlp model
  model_online_path: https://s3-us-west-2.amazonaws.com/allennlp/models/bidaf-model-
↪2017.09.15-charpad.tar.gz
  description: Pretrained model from Allennlp, for the BiDAF model (QA)
attr_file_name: null # It set, to load previously saved analysis.
group_file_name: null
rewrite_file_name: null
```

CHAPTER

FIVE

ACKNOWLEDGEMENTS

1. The design and implementation of Errudite is inspired by Allennlp.
2. We use SpaCy as the underlying preprocessing.
3. We use Altair for visualizing attributes, groups, and rewrites.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

F

G

H

I

K

L

M

O

P

Q

R

S

T

V