Snorkel

Release 0.9.7

Snorkel Team

Mar 09, 2021

PACKAGE REFERENCE

If you're looking for technical details on Snorkel's API, you're in the right place.

For more narrative walkthroughs of Snorkel fundamentals or example use cases, check out our homepage and our tutorials repo.

CHAPTER

ONE

SNORKEL ANALYSIS PACKAGE

Generic model analysis utilities shared across Snorkel.

Scorer	Calculate one or more scores from user-specified and/or user-defined metrics.
get_label_buckets	Return data point indices bucketed by label combinations.
get_label_instances	Return instances in x with the specified combination of labels.
metric_score	Evaluate a standard metric on a set of predictions/probabilities.

1.1 snorkel.analysis.Scorer

class snorkel.analysis.**Scorer**(*metrics=None*, *custom_metric_funcs=None*, *abstain_label=-1*)

Bases: object

Calculate one or more scores from user-specified and/or user-defined metrics.

Parameters

- **metrics** (Optional[List[str]]) – A list of metric names, all of which are defined in METRICS
- **custom_metric_funcs** (Optional[Mapping[str, Callable[..., float]]]) – An optional dictionary mapping the names of custom metrics to the functions that produce them. Each custom metric function should accept golds, preds, and probs as input (just like the standard metrics in METRICS) and return either a single score (float) or a dictionary of metric names to scores (if the function calculates multiple values, for example). See the unit tests for an example.
- **abstain_label** (Optional[int]) – The gold label for which examples will be ignored. By default, follow convention that abstains are -1.

Raises **ValueError** – If a specified standard metric is not found in the METRICS dictionary

metrics

A dictionary mapping metric names to the corresponding functions for calculating that metric

__init__(*metrics=None*, *custom_metric_funcs=None*, *abstain_label=-1*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([metrics, custom_metric_funcs, ...])	Initialize self.
score(golds[, preds, probs])	Calculate scores for one or more user-specified metrics.
score_slices(S, golds, preds, probs[, ...])	Calculate user-specified and/or user-defined metrics overall + slices.

score(*golds*, *preds=None*, *probs=None*)
Calculate scores for one or more user-specified metrics.

Parameters

- **golds** (ndarray) – An array of gold (int) labels to base scores on
- **preds** (Optional[ndarray]) – An [n_datapoints,] or [n_datapoints, 1] array of (int) predictions to score
- **probs** (Optional[ndarray]) – An [n_datapoints, n_classes] array of probabilistic (float) predictions
- **most metrics require either preds or probs, but not both, these** (*Because*) –
- **are optional; it is up to the metric function that will be called to** (*values*) –
- **an exception if a field it requires is not passed to the score() method.** (*raise*) –

Returns A dictionary mapping metric names to metric scores

Return type Dict[str, float]

Raises **ValueError** – If no gold labels were provided

score_slices(*S*, *golds*, *preds*, *probs*, *as_dataframe=False*)
Calculate user-specified and/or user-defined metrics overall + slices.

Parameters

- **S** (recarray) – A recarray with entries of length n_examples corresponding to slice names
- **golds** (ndarray) – Gold (aka ground truth) labels (integers)
- **preds** (ndarray) – Predictions (integers)
- **probs** (ndarray) – Probabilities (floats)
- **as_dataframe** (bool) – A boolean indicating whether to return results as pandas DataFrame (True) or dict (False)

Returns A dictionary mapping slice_name to metric names to metric scores or metrics formatted as pandas DataFrame

Return type Union[Dict, pd.DataFrame]

1.2 snorkel.analysis.get_label_buckets

snorkel.analysis.**get_label_buckets**(*y)

Return data point indices bucketed by label combinations.

Parameters ***y** – A list of np.ndarray of (int) labels

Returns A mapping of each label bucket to a NumPy array of its corresponding indices

Return type Dict[Tuple[int, ..], np.ndarray]

Example

A common use case is calling `buckets = label_buckets(Y_gold, Y_pred)` where `Y_gold` is a set of gold (i.e. ground truth) labels and `Y_pred` is a corresponding set of predicted labels.

```
>>> Y_gold = np.array([1, 1, 1, 0])
>>> Y_pred = np.array([1, 1, -1, -1])
>>> buckets = get_label_buckets(Y_gold, Y_pred)
```

The returned `buckets[(i, j)]` is a NumPy array of data point indices with true label i and predicted label j.

More generally, the returned indices within each bucket refer to the order of the labels that were passed in as function arguments.

```
>>> buckets[(1, 1)]  # true positives
array([0, 1])
>>> (1, 0) in buckets  # false positives
False
>>> (0, 1) in buckets  # false negatives
False
>>> (0, 0) in buckets  # true negatives
False
>>> buckets[(1, -1)]  # abstained positives
array([2])
>>> buckets[(0, -1)]  # abstained negatives
array([3])
```

1.3 snorkel.analysis.get_label_instances

snorkel.analysis.**get_label_instances**(*bucket*, *x*, **y*)

Return instances in x with the specified combination of labels.

Parameters

- **bucket** (Tuple[int, ...]) – A tuple of label values corresponding to which instances from x are returned
- **x** (ndarray) – NumPy array of data instances to be returned
- ***y** – A list of np.ndarray of (int) labels

Returns NumPy array of instances from x with the specified combination of labels

Return type np.ndarray

Example

A common use case is calling `get_label_instances(bucket, x.to_numpy(), Y_gold, Y_pred)` where `x` is a NumPy array of data instances that the labels correspond to, `Y_gold` is a list of gold (i.e. ground truth) labels, and `Y_pred` is a corresponding list of predicted labels.

```
>>> import pandas as pd
>>> x = pd.DataFrame(data={'col1': ["this is a string", "a second string", "a
↪third string"], 'col2': ["1", "2", "3"]})
>>> Y_gold = np.array([1, 1, 1])
>>> Y_pred = np.array([1, 0, 0])
>>> bucket = (1, 0)
```

The returned NumPy array of data instances from `x` will correspond to the rows where the first list had a 1 and the second list had a 0. >>> get_label_instances(bucket, x.to_numpy(), Y_gold, Y_pred) array([[‘a second string’, ‘2’],

[‘a third string’, ‘3’]], dtype=object)

More generally, given bucket `(i, j, ...)` and lists `y1, y2, ...` the returned data instances from `x` will correspond to the rows where y1 had label i, y2 had label j, and so on. Note that `x` and `y` must all be the same length.

1.4 snorkel.analysis.metric_score

snorkel.analysis.**metric_score**(*golds=None*, *preds=None*, *probs=None*, *metric='accuracy'*, *filter_dict=None*, ***kwargs*)

Evaluate a standard metric on a set of predictions/probabilities.

Parameters

- **golds** (Optional[ndarray]) – An array of gold (int) labels
- **preds** (Optional[ndarray]) – An array of (int) predictions
- **probs** (Optional[ndarray]) – An [n_datapoints, n_classes] array of probabilistic (float) predictions
- **metric** (str) – The name of the metric to calculate
- **filter_dict** (Optional[Dict[str, List[int]]]) – A mapping from label set name to the labels that should be filtered out for that label set

Returns The value of the requested metric

Return type float

Raises

- **ValueError** – The requested metric is not currently supported
- **ValueError** – The user attempted to calculate roc_auc score for a non-binary problem

CHAPTER

TWO

SNORKEL AUGMENTATION PACKAGE

Programmatic data set augmentation: TF creation and data generation utilities.

ApplyAllPolicy	Apply all TFs in order to each data point.
ApplyEachPolicy	Apply each TF individually to each data point.
ApplyOnePolicy	Apply a single TF to each data point.
MeanFieldPolicy	Sample sequences of TFs according to a distribution.
PandasTFApplier	TF applier for a Pandas DataFrame.
RandomPolicy	Naive random augmentation policy.
TFApplier	TF applier for a list of data points.
TransformationFunction	Base class for TFs.
transformation_function	Decorate functions to create TFs.

2.1 snorkel.augmentation.ApplyAllPolicy

class snorkel.augmentation.**ApplyAllPolicy**(*n_tfs*, *n_per_original=1*, *keep_original=True*)

Bases: snorkel.augmentation.policy.core.Policy

Apply all TFs in order to each data point.

While this can be used as a baseline policy, using a random policy is more standard. See RandomPolicy.

Parameters

- **n_tfs** (int) – Total number of TFs
- **n_per_original** (int) – Number of transformed data points for each original data point
- **keep_original** (bool) – Keep untransformed data point in augmented data set? Note that even if in-place modifications are made to the original data point by the TFs being applied, the original data point will remain unchanged.

Example

```
>>> policy = ApplyAllPolicy(3, n_per_original=2, keep_original=False)
>>> policy.generate_for_example()
[[0, 1, 2], [0, 1, 2]]
```

n

Total number of TFs

n_per_original
See above

keep_original
See above

__init__ (*n_tfs*, *n_per_original=1*, *keep_original=True*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(n_tfs[, n_per_original, keep_original])	Initialize self.
generate()	Generate indices of all TFs in order.
generate_for_example()	Generate all sequences of TF indices for a single example.

generate()
Generate indices of all TFs in order.

Returns Indices of all TFs in order.

Return type List[int]

generate_for_example()
Generate all sequences of TF indices for a single example.

Generates *n_per_original* sequences, and adds an empty sequence if *keep_original* is True.

Returns Sequences of indices of TFs to run on data point in order.

Return type List[List[int]]

2.2 snorkel.augmentation.ApplyEachPolicy

class snorkel.augmentation.**ApplyEachPolicy**(*n_tfs*, *keep_original=True*)
Bases: snorkel.augmentation.policy.core.Policy

Apply each TF individually to each data point.

This can be used as a baseline policy when using complex transformations which might degenerate if combined.

Parameters

- **n_tfs** (int) – Total number of TFs
- **keep_original** (bool) – Keep untransformed data point in augmented data set? Note that even if in-place modifications are made to the original data point by the TFs being applied, the original data point will remain unchanged.

Example

```
>>> policy = ApplyEachPolicy(3, keep_original=True)
>>> policy.generate_for_example()
[[], [0], [1], [2]]
```

n
Total number of TFs

n_per_original
Total number of TFs

keep_original
See above

__init__(*n_tfs*, *keep_original=True*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(n_tfs[, keep_original])	Initialize self.
generate()	Generate a sequence of TF indices.
generate_for_example()	Generate all length-one sequences for a single example.

generate()
Generate a sequence of TF indices.

Returns Indices of TFs to run on data point in order.

Return type List[int]

Raises **NotImplementedError** – Subclasses need to implement this method

generate_for_example()
Generate all length-one sequences for a single example.

Returns Sequences of indices of TFs to run on data point in order.

Return type List[List[int]]

2.3 snorkel.augmentation.ApplyOnePolicy

class snorkel.augmentation.**ApplyOnePolicy**(*n_per_original=1*, *keep_original=True*)
Bases: snorkel.augmentation.policy.core.ApplyAllPolicy

Apply a single TF to each data point.

__init__(*n_per_original=1*, *keep_original=True*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([n_per_original, keep_original])	Initialize self.
generate()	Generate indices of all TFs in order.
generate_for_example()	Generate all sequences of TF indices for a single example.

generate()
Generate indices of all TFs in order.

Returns Indices of all TFs in order.

Return type List[int]

generate_for_example()
Generate all sequences of TF indices for a single example.

Generates *n_per_original* sequences, and adds an empty sequence if *keep_original* is True.

Returns Sequences of indices of TFs to run on data point in order.

Return type List[List[int]]

2.4 snorkel.augmentation.MeanFieldPolicy

class snorkel.augmentation.**MeanFieldPolicy**(*n_tfs*, *sequence_length=1*, *p=None*, *n_per_original=1*, *keep_original=True*)
Bases: snorkel.augmentation.policy.core.Policy

Sample sequences of TFs according to a distribution.

Samples sequences of indices of a specified length from a user-provided distribution. A distribution over TFs can be learned by a TANDA mean-field model, for example. See https://hazyresearch.github.io/snorkel/blog/tanda.html

Parameters

- **n_tfs** (int) – Total number of TFs
- **sequence_length** (int) – Number of TFs to run on each data point
- **p** (Optional[Sequence[float]]) – Probability distribution from which to sample TF indices. Must have length n_tfs and be a valid distribution.
- **n_per_original** (int) – Number of transformed data points per original
- **keep_original** (bool) – Keep untransformed data point in augmented data set? Note that even if in-place modifications are made to the original data point by the TFs being applied, the original data point will remain unchanged.

n
Total number of TFs

n_per_original
See above

keep_original
See above

sequence_length
See above

__init__(*n_tfs*, *sequence_length=1*, *p=None*, *n_per_original=1*, *keep_original=True*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(n_tfs[, sequence_length, p, ...])	Initialize self.
generate()	Generate a sequence of TF indices by sampling from distribution.
generate_for_example()	Generate all sequences of TF indices for a single example.

generate()
Generate a sequence of TF indices by sampling from distribution.

Returns Indices of TFs to run on data point in order.

Return type List[int]

generate_for_example()
Generate all sequences of TF indices for a single example.

Generates *n_per_original* sequences, and adds an empty sequence if *keep_original* is True.

Returns Sequences of indices of TFs to run on data point in order.

Return type List[List[int]]

2.5 snorkel.augmentation.PandasTFApplier

class snorkel.augmentation.**PandasTFApplier**(*tfs*, *policy*)
Bases: snorkel.augmentation.apply.core.BaseTFApplier

TF applier for a Pandas DataFrame.

Data points are stored as Series in a DataFrame. The TFs run on data points obtained via a pandas.DataFrame.iterrows call, which is single-process and can be slow for large DataFrames. For large datasets, consider DaskTFApplier or SparkTFApplier.

__init__(*tfs*, *policy*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(tfs, policy)	Initialize self.
apply(df[, progress_bar])	Augment a Pandas DataFrame of data points using TFs and policy.
apply_generator(df, batch_size)	Augment a Pandas DataFrame of data points using TFs and policy in batches.

apply(*df*, *progress_bar=True*)
Augment a Pandas DataFrame of data points using TFs and policy.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data points to be transformed
- **progress_bar** (bool) – Display a progress bar?

Returns Pandas DataFrame of data points in augmented data set

Return type pd.DataFrame

apply_generator(*df*, *batch_size*)
Augment a Pandas DataFrame of data points using TFs and policy in batches.

This method acts as a generator, yielding augmented data points for a given input batch of data points. This can be useful in a training loop when it is too memory-intensive to pregenerate all transformed examples.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data points to be transformed
- **batch_size** (int) – Batch size for generator. Yields augmented data points for the next batch_size input data points.

Returns Pandas DataFrame of data points in augmented data set

Return type pd.DataFrame

2.6 snorkel.augmentation.RandomPolicy

class snorkel.augmentation.**RandomPolicy**(*n_tfs*, *sequence_length=1*, *n_per_original=1*, *keep_original=True*)
Bases: snorkel.augmentation.policy.sampling.MeanFieldPolicy

Naive random augmentation policy.

Samples sequences of TF indices a specified length at random from the total number of TFs. Sampling uniformly at random is a common baseline approach to data augmentation.

Parameters

- **n_tfs** (int) – Total number of TFs
- **sequence_length** (int) – Number of TFs to run on each data point
- **n_per_original** (int) – Number of transformed data points per original
- **keep_original** (bool) – Keep untransformed data point in augmented data set? Note that even if in-place modifications are made to the original data point by the TFs being applied, the original data point will remain unchanged.

n
Total number of TFs

n_per_original
See above

keep_original
See above

sequence_length
See above

__init__(*n_tfs*, *sequence_length=1*, *n_per_original=1*, *keep_original=True*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(n_tfs[, sequence_length, ...])	Initialize self.
generate()	Generate a sequence of TF indices by sampling from distribution.
generate_for_example()	Generate all sequences of TF indices for a single example.

generate()
Generate a sequence of TF indices by sampling from distribution.

Returns Indices of TFs to run on data point in order.

Return type List[int]

generate_for_example()
Generate all sequences of TF indices for a single example.

Generates *n_per_original* sequences, and adds an empty sequence if *keep_original* is True.

Returns Sequences of indices of TFs to run on data point in order.

Return type List[List[int]]

2.7 snorkel.augmentation.TFApplier

class snorkel.augmentation.**TFApplier**(*tfs*, *policy*)
Bases: snorkel.augmentation.apply.core.BaseTFApplier

TF applier for a list of data points.

Augments a list of data points (e.g. SimpleNamespace). Primarily useful for testing.

__init__(*tfs*, *policy*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(tfs, policy)	Initialize self.
apply(data_points[, progress_bar])	Augment a list of data points using TFs and policy.
apply_generator(data_points, batch_size)	Augment a list of data points using TFs and policy in batches.

apply(*data_points*, *progress_bar=True*)
Augment a list of data points using TFs and policy.

Parameters

- **data_points** (Sequence[Any]) – List containing data points to be transformed
- **progress_bar** (bool) – Display a progress bar?

Returns List of data points in augmented data set

Return type List[DataPoint]

apply_generator(*data_points*, *batch_size*)
Augment a list of data points using TFs and policy in batches.

This method acts as a generator, yielding augmented data points for a given input batch of data points. This can be useful in a training loop when it is too memory-intensive to pregenerate all transformed examples.

Parameters

- **data_points** (Sequence[Any]) – List containing data points to be transformed
- **batch_size** (int) – Batch size for generator. Yields augmented data points for the next batch_size input data points.

Yields *List[DataPoint]* – List of data points in augmented data set for batches of inputs

Return type Iterator[List[Any]]

2.8 snorkel.augmentation.TransformationFunction

class snorkel.augmentation.**TransformationFunction**(*name*, *field_names=None*, *mapped_field_names=None*, *pre=None*, *memoize=False*, *memoize_key=None*)

Bases: snorkel.map.core.Mapper

Base class for TFs.

See snorkel.map.core.Mapper for details.

__init__(*name*, *field_names=None*, *mapped_field_names=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name[, field_names, ...])	Initialize self.
reset_cache()	Reset the memoization cache.
run(**kwargs)	Run the mapping operation using the input fields.

__call__(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If memoize is set to True, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (Any) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type None

run (***kwargs*)
Run the mapping operation using the input fields.

The inputs to this function are fed by extracting the fields of the input data point using the keys of `field_names`. The output field names are converted using `mapped_field_names` and added to the data point.

Returns A mapping from canonical output field names to their values.

Return type Optional[FieldMap]

Raises **NotImplementedError** – Subclasses must implement this method

2.9 snorkel.augmentation.transformation_function

class snorkel.augmentation.**transformation_function**(*name=None*, *pre=None*, *memoize=False*, *memoize_key=None*)

Bases: `snorkel.map.core.lambda_mapper`

Decorate functions to create TFs.

See `snorkel.map.core.lambda_mapper` for details.

Example

```
>>> @transformation_function()
... def square(x):
...     x.num = x.num ** 2
...     return x
>>> from types import SimpleNamespace
>>> square(SimpleNamespace(num=2))
namespace(num=4)
```

__init__ (*name=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, pre, memoize, memoize_key])	Initialize self.

__call__ (*f*)
Wrap a function to create a `LambdaMapper`.

Parameters **f** (`Callable`[[`Any`], `Optional`[`Any`]]) – Function executing the mapping operation

Returns New `LambdaMapper` executing operation in wrapped function

Return type *LambdaMapper*

CHAPTER

THREE

SNORKEL CLASSIFICATION PACKAGE

PyTorch-based multi-task learning framework for discriminative modeling.

Checkpointer	Manager for checkpointing model.
CheckpointerConfig	Manager for checkpointing model.
DictDataLoader	A DataLoader that uses the appropriate collate_fn for a `DictDataset`.
DictDataset	A dataset where both the data fields and labels are stored in as dictionaries.
LogManager	A class to manage logging during training progress.
LogManagerConfig	Manager for checkpointing model.
LogWriter	A class for writing logs.
LogWriterConfig	Manager for checkpointing model.
MultitaskClassifier	A classifier built from one or more tasks to support advanced workflows.
Operation	A single operation (forward pass of a module) to execute in a Task.
Task	A single task (a collection of modules and specified path through them).
TensorBoardWriter	A class for logging to Tensorboard during training process.
Trainer	A class for training a MultitaskClassifier.
cross_entropy_with_probs	Calculate cross-entropy loss when targets are probabilities (floats), not ints.

3.1 snorkel.classification.Checkpointer

class snorkel.classification.**Checkpointer**(*counter_unit*, *evaluation_freq*, ***kwargs*)

Bases: `object`

Manager for checkpointing model.

Parameters

- **counter_unit** (`str`) – The unit to use when determining when its time to checkpoint (one of ["epochs", "batches", "points"]); must match the counter_unit of LogManager
- **evaluation_freq** (`float`) – How frequently the model is being evaluated (this is the maximum frequency that checkpointing can occur, which will happen if checkpoint_factor==1)
- **kwargs** (`Any`) – Config merged with `default_config["checkpointer_config"]`

__init__ (*counter_unit*, *evaluation_freq*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(counter_unit, evaluation_freq, **kwargs)	Initialize self.
checkpoint(iteration, model, metric_dict)	Check if iteration and current metrics necessitate a checkpoint.
clear()	Clear existing checkpoint files, besides the best-yet model.
load_best_model(model)	Load the best model from the checkpoint.

checkpoint (*iteration*, *model*, *metric_dict*)
Check if iteration and current metrics necessitate a checkpoint.

Parameters

- **iteration** (float) – Current training iteration
- **model** (MultitaskClassifier) – Model to checkpoint
- **metric_dict** (Dict[str, float]) – Current performance metrics for model

Return type None

clear()
Clear existing checkpoint files, besides the best-yet model.

Return type None

load_best_model (*model*)
Load the best model from the checkpoint.

Return type MultitaskClassifier

3.2 snorkel.classification.CheckpointerConfig

class snorkel.classification.**CheckpointerConfig**
Bases: tuple

Manager for checkpointing model.

Parameters

- **checkpoint_dir** – The path to a directory where checkpoints will be saved The Trainer will set this to the log directory if it is None
- **checkpoint_factor** – Check for a best model every this many evaluations. For example, if evaluation_freq is 0.5 epochs and checkpoint_factor is 2, then checkpointing will be attempted every 1 epochs.
- **checkpoint_metric** – The metric to checkpoint on, of the form "task/dataset/split/metric:mode" where mode is "min" or "max".

- **checkpoint_task_metrics** – Additional metrics to save best models for. Note that the best model according to *checkpoint_metric* will be the one that is loaded after training and used for early stopping.
- **checkpoint_runway** – No checkpointing will occur for the first this many checkpoint_units
- **checkpoint_clear** – If True, clear all checkpoints besides the best so far.

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
count(value)	
index(value, [start, [stop]])	Raises ValueError if the value is not present.

Attributes

checkpoint_clear	Alias for field number 5
checkpoint_dir	Alias for field number 0
checkpoint_factor	Alias for field number 1
checkpoint_metric	Alias for field number 2
checkpoint_runway	Alias for field number 4
checkpoint_task_metrics	Alias for field number 3

property checkpoint_clear
Alias for field number 5

property checkpoint_dir
Alias for field number 0

property checkpoint_factor
Alias for field number 1

property checkpoint_metric
Alias for field number 2

property checkpoint_runway
Alias for field number 4

property checkpoint_task_metrics
Alias for field number 3

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

3.3 snorkel.classification.DictDataLoader

class snorkel.classification.**DictDataLoader**(*dataset*, *collate_fn=<function collate_dicts>*, ***kwargs*)
Bases: torch.utils.data.dataloader.DataLoader

A DataLoader that uses the appropriate collate_fn for a `DictDataset`.

Parameters

- **dataset** (`DictDataset`) – A dataset to wrap
- **collate_fn** (`Callable[..., Any]`) – The collate function to use when combining multiple indexed examples for a single batch. Usually the default collate_dicts() method should be used, but it can be overriden if you want to use different collate logic.
- **kwargs** (`Any`) – Keyword arguments to pass on to DataLoader.__init__()

__init__ (*dataset*, *collate_fn=<function collate_dicts>*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type `None`

Methods

__init__(dataset[, collate_fn])	Initialize self.
check_worker_number_rationality()	

Attributes

multiprocessing_context

3.4 snorkel.classification.DictDataset

class snorkel.classification.**DictDataset**(*name*, *split*, *X_dict*, *Y_dict*)
Bases: `torch.utils.data.dataset.Dataset`

A dataset where both the data fields and labels are stored in as dictionaries.

Parameters

- **name** (`str`) – The name of the dataset (e.g., this will be used to report metrics on a per-dataset basis)
- **split** (`str`) – The name of the split that the data in this object represents
- **X_dict** (`Dict[str, Any]`) – A map from field name to values (e.g., {"tokens": ..., "uids": ...})
- **Y_dict** (`Dict[str, Tensor]`) – A map from task name to its corresponding set of labels

Raises **ValueError** – All values in the `Y_dict` must be of type torch.Tensor

name
See above

split
See above

X_dict
See above

Y_dict
See above

__init__ (*name*, *split*, *X_dict*, *Y_dict*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, split, X_dict, Y_dict)	Initialize self.
from_tensors(X_tensor, Y_tensor, split[, ...])	Initialize a DictDataset from PyTorch Tensors.

classmethod from_tensors (*X_tensor*, *Y_tensor*, *split*, *input_data_key='input_data'*, *task_name='task'*, *dataset_name='SnorkelDataset'*)
Initialize a DictDataset from PyTorch Tensors.

Parameters

- **X_tensor** (Tensor) – Input data of shape [num_examples, feature_dim]
- **Y_tensor** (Tensor) – Labels of shape [num_samples, num_classes]
- **split** (str) – Name of data split corresponding to this dataset.
- **input_data_key** (str) – Name of data field to initialize in X_dict
- **task_name** (str) – Name of task and corresponding label key in Y_dict
- **dataset_name** (str) – Name of DictDataset to be initialized; See __init__ above.

Returns Class initialized with single task and label corresponding to input data

Return type *DictDataset*

3.5 snorkel.classification.LogManager

class snorkel.classification.**LogManager** (*n_batches_per_epoch*, *log_writer=None*, *checkpointer=None*, ***kwargs*)
Bases: object

A class to manage logging during training progress.

Parameters

- **n_batches_per_epoch** (int) – Total number batches per epoch
- **log_writer** (Optional[LogWriter]) – LogWriter for current run logs
- **checkpointer** (Optional[Checkpointer]) – Checkpointer for current model
- **kwargs** (Any) – Settings to update in LogManagerConfig

__init__ (*n_batches_per_epoch*, *log_writer=None*, *checkpointer=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(n_batches_per_epoch[, log_writer, ...])	Initialize self.
cleanup(model)	Close the log writer and checkpointer if needed.
reset()	Reset counters.
trigger_checkpointing()	Check if current counts trigger checkpointing.
trigger_evaluation()	Check if current counts trigger evaluation.
update(batch_size)	Update the count and total number.

cleanup(*model*)
Close the log writer and checkpointer if needed. Reload best model.

Return type `MultitaskClassifier`

reset()
Reset counters.

Return type `None`

trigger_checkpointing()
Check if current counts trigger checkpointing.

Return type `bool`

trigger_evaluation()
Check if current counts trigger evaluation.

Return type `bool`

update(*batch_size*)
Update the count and total number.

Return type `None`

3.6 snorkel.classification.LogManagerConfig

class snorkel.classification.**LogManagerConfig**
Bases: `tuple`

Manager for checkpointing model.

Parameters

- **counter_unit** – The unit to use when assessing when it's time to log. Options are ["epochs", "batches", "points"]
- **evaluation_freq** – Evaluate performance on the validation set every this many counter_units

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
count(value)	
index(value, [start, [stop]])	Raises ValueError if the value is not present.

Attributes

counter_unit	Alias for field number 0
evaluation_freq	Alias for field number 1

count(*value*) → integer – return number of occurrences of value

property counter_unit
Alias for field number 0

property evaluation_freq
Alias for field number 1

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

3.7 snorkel.classification.LogWriter

class snorkel.classification.**LogWriter**(***kwargs*)
Bases: object

A class for writing logs.

Parameters **kwargs** (Any) – Settings to merge into LogWriterConfig

config
Merged configuration

run_name
Name of run if provided, otherwise date-time combination

log_dir
The root directory where logs should be saved

run_log
Dictionary of scalar values to log, keyed by value name

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(**kwargs)	Initialize self.
add_scalar(name, value, step)	Log a scalar variable.
cleanup()	Perform final operations and close writer if necessary.
write_config(config[, config_filename])	Dump the config to file.
write_json(dict_to_write, filename)	Dump a JSON-compatbile object to root log directory.
write_log(log_filename)	Dump the scalar value log to file.
write_text(text, filename)	Dump user-provided text to filename (e.g., the launch command).

add_scalar(*name*, *value*, *step*)
Log a scalar variable.

Parameters

- **name** (str) – Name of the scalar collection
- **value** (float) – Value of scalar
- **step** (float) – Step axis value

Return type None

cleanup()
Perform final operations and close writer if necessary.

Return type None

write_config(*config*, *config_filename='config.json'*)
Dump the config to file.

Parameters

- **config** (Namedtuple) – JSON-compatible config to write to file
- **config_filename** (str) – Name of file in logging directory to write to

Return type None

write_json(*dict_to_write*, *filename*)
Dump a JSON-compatbile object to root log directory.

Parameters

- **dict_to_write** (Mapping[str, Any]) – JSON-compatbile object to log
- **filename** (str) – Name of file in logging directory to write to

Return type None

write_log(*log_filename*)
Dump the scalar value log to file.

Parameters **log_filename** (str) – Name of file in logging directory to write to

Return type None

write_text(*text*, *filename*)
Dump user-provided text to filename (e.g., the launch command).

Parameters

- **text** (str) – Text to write
- **filename** (str) – Name of file in logging directory to write to

Return type None

3.8 snorkel.classification.LogWriterConfig

class snorkel.classification.**LogWriterConfig**
Bases: tuple

Manager for checkpointing model.

Parameters

- **log_dir** – The root directory where logs should be saved
- **run_name** – The name of this particular run (defaults to date-time combination if None)

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__	Initialize self.
count(value)	
index(value, [start, [stop]])	Raises ValueError if the value is not present.

Attributes

log_dir	Alias for field number 0
run_name	Alias for field number 1

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

property log_dir
Alias for field number 0

property run_name
Alias for field number 1

3.9 snorkel.classification.MultitaskClassifier

class snorkel.classification.**MultitaskClassifier**(*tasks*, *name=None*, ***kwargs*)
Bases: torch.nn.modules.module.Module

A classifier built from one or more tasks to support advanced workflows.

Parameters

- **tasks** (List[Task]) – A list of Tasks to build a model from
- **name** (Optional[str]) – The name of the classifier

config
The config dict containing the settings for this model

name
See above

module_pool
A dictionary of all modules used by any of the tasks (See Task docstring)

task_names
See Task docstring

op_sequences
See Task docstring

loss_funcs
See Task docstring

output_funcs
See Task docstring

scorers
See Task docstring

__init__(*tasks*, *name=None*, ***kwargs*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

Return type None

Methods

__init__(tasks[, name])	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
add_task(task)	Add a single task to the network.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
calculate_loss(X_dict, Y_dict)	Calculate the loss for each task and the number of data points contributing.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(X_dict, task_names)	Do a forward pass through the network for all specified tasks.
half()	Casts all floating point parameters and buffers to half datatype.
load(model_path)	Load a saved model from the provided file path and moves it to a device.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.

Continued on next page

Table 14 – continued from previous page

named_modules([memo, prefix])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
predict(dataloader[, return_preds, remap_labels])	Calculate probabilities, (optionally) predictions, and pull out gold labels.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
save(model_path)	Save the model to the specified file path.
score(dataloaders[, remap_labels, as_dataframe])	Calculate scores for the provided DictDataLoaders.
share_memory() **rtype** ~T	
state_dict([destination, prefix, keep_vars])	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination
dump_patches

add_task(*task*)
Add a single task to the network.

Parameters **task** (Task) – A Task to add

Return type None

calculate_loss(*X_dict*, *Y_dict*)
Calculate the loss for each task and the number of data points contributing.

Parameters

- **X_dict** (Dict[str, Any]) – A dict of data fields
- **Y_dict** (Dict[str, Tensor]) – A dict from task names to label sets

Returns A dict of losses by task name and seen examples by task name

Return type Dict[str, torch.Tensor], Dict[str, float]

forward(*X_dict*, *task_names*)
Do a forward pass through the network for all specified tasks.

Parameters

- **X_dict** (Dict[str, Any]) – A dict of data fields
- **task_names** (Iterable[str]) – The names of the tasks to execute the forward pass for

Returns A dict mapping each operation name to its corresponding output

Return type OutputDict

Raises

- **TypeError** – If an Operation input has an invalid type
- **ValueError** – If a specified Operation failed to execute

load(*model_path*)
Load a saved model from the provided file path and moves it to a device.

Parameters **model_path** (str) – The path to a saved model

Return type None

predict(*dataloader*, *return_preds=False*, *remap_labels={}*)
Calculate probabilities, (optionally) predictions, and pull out gold labels.

Parameters

- **dataloader** (DictDataLoader) – A DictDataLoader to make predictions for
- **return_preds** (bool) – If True, include predictions in the return dict (not just probabilities)
- **remap_labels** (Dict[str, Optional[str]]) – A dict specifying which labels in the dataset's Y_dict (key) to remap to a new task (value)

Returns A dictionary mapping label type ('golds', 'probs', 'preds') to values

Return type Dict[str, Dict[str, torch.Tensor]]

save(*model_path*)
Save the model to the specified file path.

Parameters **model_path** (str) – The path where the model should be saved

Raises **BaseException** – If the torch.save() method fails

Return type None

score(*dataloaders*, *remap_labels={}*, *as_dataframe=False*)
Calculate scores for the provided DictDataLoaders.

Parameters

- **dataloaders** (List[DictDataLoader]) – A list of DictDataLoaders to calculate scores for
- **remap_labels** (Dict[str, Optional[str]]) – A dict specifying which labels in the dataset's Y_dict (key) to remap to a new task (value)
- **as_dataframe** (bool) – A boolean indicating whether to return results as pandas DataFrame (True) or dict (False)

Returns A dictionary mapping metric names to corresponding scores Metric names will be of the form "task/dataset/split/metric"

Return type Dict[str, float]

3.10 snorkel.classification.Operation

class snorkel.classification.**Operation**(*module_name*, *inputs*, *name=None*)

Bases: object

A single operation (forward pass of a module) to execute in a Task.

See Task for more detail on the usage and semantics of an Operation.

Parameters

- **name** (Optional[str]) – The name of this operation (defaults to module_name since for most workflows, each module is only used once per forward pass)
- **module_name** (str) – The name of the module in the module pool that this operation uses
- **inputs** (Sequence[Union[str, Tuple[str, str]]]) – The inputs that the specified module expects, given as a list of names of previous operations (or optionally a tuple of the operation name and a key if the output of that module is a dict instead of a Tensor). Note that the original input to the model can be referred to as "_input_".

Example

```
>>> op1 = Operation(module_name="linear1", inputs=[("_input_", "features")])
>>> op2 = Operation(module_name="linear2", inputs=["linear1"])
>>> op_sequence = [op1, op2]
```

name

See above

module_name

See above

inputs

See above

__init__(*module_name*, *inputs*, *name=None*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(module_name, inputs[, name])	Initialize self.

3.11 snorkel.classification.Task

class snorkel.classification.**Task**(*name*, *module_pool*, *op_sequence*, *scorer=<snorkel.analysis.scorer.Scorer object>*, *loss_func=None*, *output_func=None*)

Bases: object

A single task (a collection of modules and specified path through them).

Parameters

- **name** (str) – The name of the task
- **module_pool** (ModuleDict) – A ModuleDict mapping module names to the modules themselves
- **op_sequence** (Sequence[Operation]) – A list of Operations to execute in order, defining the flow of information through the network for this task
- **scorer** (Scorer) – A Scorer with the desired metrics to calculate for this task
- **loss_func** (Optional[Callable[..., Tensor]]) – A function that converts final logits into loss values. Defaults to F.cross_entropy() if none is provided. To use probalistic labels for training, use the Snorkel-defined method cross_entropy_with_probs() instead.
- **output_func** (Optional[Callable[..., Tensor]]) – A function that converts final logits into 'outputs' (e.g. probabilities) Defaults to F.softmax(..., dim=1).

name
See above

module_pool
See above

op_sequence
See above

scorer
See above

loss_func
See above

output_func
See above

__init__(*name*, *module_pool*, *op_sequence*, *scorer=<snorkel.analysis.scorer.Scorer object>*, *loss_func=None*, *output_func=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, module_pool, op_sequence[, ...])	Initialize self.

3.12 snorkel.classification.TensorBoardWriter

class snorkel.classification.**TensorBoardWriter**(***kwargs*)

Bases: snorkel.classification.training.loggers.log_writer.LogWriter

A class for logging to Tensorboard during training process.

See LogWriter for more attributes.

Parameters **kwargs** (Any) – Passed to LogWriter initializer

writer

SummaryWriter for logging and visualization

__init__(***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(**kwargs)	Initialize self.
add_scalar(name, value, step)	Log a scalar variable to TensorBoard.
cleanup()	Close the SummaryWriter.
write_config(config[, config_filename])	Dump the config to file and add it to TensorBoard.
write_json(dict_to_write, filename)	Dump a JSON-compatbile object to root log directory.
write_log(log_filename)	Dump the scalar value log to file.
write_text(text, filename)	Dump user-provided text to filename (e.g., the launch command).

add_scalar(*name*, *value*, *step*)

Log a scalar variable to TensorBoard.

Parameters

- **name** (str) – Name of the scalar collection
- **value** (float) – Value of scalar
- **step** (float) – Step axis value

Return type None

cleanup()

Close the SummaryWriter.

Return type None

write_config(*config*, *config_filename='config.json'*)

Dump the config to file and add it to TensorBoard.

Parameters

- **config** (Namedtuple) – JSON-compatible config to write to TensorBoard
- **config_filename** (str) – File to write config to

Return type None

write_json(*dict_to_write*, *filename*)
Dump a JSON-compatbile object to root log directory.

Parameters

- **dict_to_write** (Mapping[str, Any]) – JSON-compatbile object to log
- **filename** (str) – Name of file in logging directory to write to

Return type None

write_log(*log_filename*)
Dump the scalar value log to file.

Parameters **log_filename** (str) – Name of file in logging directory to write to

Return type None

write_text(*text*, *filename*)
Dump user-provided text to filename (e.g., the launch command).

Parameters

- **text** (str) – Text to write
- **filename** (str) – Name of file in logging directory to write to

Return type None

3.13 snorkel.classification.Trainer

class snorkel.classification.**Trainer**(*name=None*, ***kwargs*)
Bases: object

A class for training a MultitaskClassifier.

Parameters

- **name** (Optional[str]) – An optional name for this trainer object
- **kwargs** (Any) – Settings to be merged into the default Trainer config dict

name
See above

config
The config dict with the settings for the Trainer

checkpointer
Saves the best model seen during training

log_manager
Identifies when its time to log or evaluate on the valid set

log_writer
Writes training statistics to file or TensorBoard

optimizer
Updates model weights based on the loss

lr_scheduler
Adjusts the learning rate over the course of training

batch_scheduler
Returns batches from the DataLoaders in a particular order for training

__init__(*name=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name])	Initialize self.
fit(model, dataloaders)	Train a MultitaskClassifier.
load(trainer_path, model)	Load trainer config and optimizer state from the specified json file path to the trainer object.
save(trainer_path)	Save the trainer config to the specified file path in json format.

fit(*model*, *dataloaders*)
Train a MultitaskClassifier.

Parameters

- **model** (MultitaskClassifier) – The model to train
- **dataloaders** (List[DictDataLoader]) – A list of DataLoaders. These will split into train, valid, and test splits based on the split attribute of the DataLoaders.

Return type None

load(*trainer_path*, *model*)
Load trainer config and optimizer state from the specified json file path to the trainer object. The optimizer state is stored, too. However, it only makes sense if loaded with the correct model again.

Parameters

- **trainer_path** (str) – The path to the saved trainer config to be loaded
- **model** (Optional[MultitaskClassifier]) – MultitaskClassifier for which the optimizer has been set. Parameters of optimizer must fit to model parameters. This model shall be the model which was fit by the stored Trainer.

Example

Saving model and corresponding trainer: >>> model.save('./my_saved_model_file') # doctest: +SKIP >>> trainer.save('./my_saved_trainer_file') # doctest: +SKIP Now we can resume training and load the saved model and trainer into new model and trainer objects: >>> new_model.load('./my_saved_model_file') # doctest: +SKIP >>> new_trainer.load('./my_saved_trainer_file', model=new_model) # doctest: +SKIP >>> new_trainer.fit(...) # doctest: +SKIP

Return type None

save(*trainer_path*)
Save the trainer config to the specified file path in json format.

Parameters **trainer_path** (str) – The path where trainer config and optimizer state should be saved.

Return type None

3.14 snorkel.classification.cross_entropy_with_probs

snorkel.classification.**cross_entropy_with_probs**(*input*, *target*, *weight=None*, *reduction='mean'*)

Calculate cross-entropy loss when targets are probabilities (floats), not ints.

PyTorch's F.cross_entropy() method requires integer labels; it does accept probabilistic labels. We can, however, simulate such functionality with a for loop, calculating the loss contributed by each class and accumulating the results. Libraries such as keras do not require this workaround, as methods like "categorical_crossentropy" accept float labels natively.

Note that the method signature is intentionally very similar to F.cross_entropy() so that it can be used as a drop-in replacement when target labels are changed from from a 1D tensor of ints to a 2D tensor of probabilities.

Parameters

- **input** (Tensor) – A [num_points, num_classes] tensor of logits
- **target** (Tensor) – A [num_points, num_classes] tensor of probabilistic target labels
- **weight** (Optional[Tensor]) – An optional [num_classes] array of weights to multiply the loss by per class
- **reduction** (str) – One of "none", "mean", "sum", indicating whether to return one loss per data point, the mean loss, or the sum of losses

Returns The calculated loss

Return type torch.Tensor

Raises **ValueError** – If an invalid reduction keyword is submitted

CHAPTER

FOUR

SNORKEL LABELING PACKAGE

Programmatic data set labeling: LF creation, models, and analysis utilities.

apply.dask.DaskLFApplier	LF applier for a Dask DataFrame.
LFAnalysis	Run analyses on LFs using label matrix.
LFApplier	LF applier for a list of data points (e.g.
model.label_model.LabelModel	A model for learning the LF accuracies and combining their output labels.
LabelingFunction	Base class for labeling functions.
model.baselines.MajorityClassVoter	Majority class label model.
model.baselines.MajorityLabelVoter	Majority vote label model.
lf.nlp.NLPLabelingFunction	Special labeling function type for spaCy-based LFs.
PandasLFApplier	LF applier for a Pandas DataFrame.
apply.dask.PandasParallelLFApplier	Parallel LF applier for a Pandas DataFrame.
model.baselines.RandomVoter	Random vote label model.
apply.spark.SparkLFApplier	LF applier for a Spark RDD.
lf.nlp_spark.SparkNLPLabelingFunction	Special labeling function type for SpaCy-based LFs running on Spark.
filter_unlabeled_dataframe	Filter out examples not covered by any labeling function.
labeling_function	Decorator to define a LabelingFunction object from a function.
lf.nlp.nlp_labeling_function	Decorator to define an NLPLabelingFunction object from a function.
lf.nlp_spark.spark_nlp_labeling_functio	Decorator to define a SparkNLPLabelingFunction object from a function.

4.1 snorkel.labeling.apply.dask.DaskLFApplier

class snorkel.labeling.apply.dask.**DaskLFApplier**(*lfs*)

Bases: snorkel.labeling.apply.core.BaseLFApplier

LF applier for a Dask DataFrame.

Dask DataFrames consist of partitions, each being a Pandas DataFrame. This allows for efficient parallel computation over DataFrame rows. For more information, see https://docs.dask.org/en/stable/dataframe.html

__init__(*lfs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(df[, scheduler, fault_tolerant])	Label Dask DataFrame of data points with LFs.

apply (*df*, *scheduler='processes'*, *fault_tolerant=False*)
Label Dask DataFrame of data points with LFs.

Parameters

- **df** (*dask.dataframe.DataFrame*) – Dask DataFrame containing data points to be labeled by LFs
- **scheduler** (Union[str, dask.distributed.Client]) – A Dask scheduling configuration: either a string option or a Client. For more information, see https://docs.dask.org/en/stable/scheduling.html#
- **fault_tolerant** (bool) – Output -1 if LF execution fails?

Returns Matrix of labels emitted by LFs

Return type np.ndarray

4.2 snorkel.labeling.LFAnalysis

class snorkel.labeling.**LFAnalysis** (*L*, *lfs=None*)
Bases: object

Run analyses on LFs using label matrix.

Parameters

- **L** (ndarray) – Label matrix where L_{i,j} is the label given by the jth LF to the ith candidate (using -1 for abstain)
- **lfs** (Optional[List[LabelingFunction]]) – Labeling functions used to generate L

Raises **ValueError** – If number of LFs and number of LF matrix columns differ

L
See above.

__init__ (*L*, *lfs=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(L[, lfs])	Initialize self.
label_conflict()	Compute the fraction of data points with conflicting (non-abstain) labels.
label_coverage()	Compute the fraction of data points with at least one label.

Continued on next page

Table 3 – continued from previous page

label_overlap()	Compute the fraction of data points with at least two (non-abstain) labels.
lf_conflicts([normalize_by_overlaps])	Compute frac.
lf_coverages()	Compute frac.
lf_empirical_accuracies(Y)	Compute empirical accuracy against a set of labels Y for each LF.
lf_empirical_probs(Y, k)	Estimate conditional probability tables for each LF.
lf_overlaps([normalize_by_coverage])	Compute frac.
lf_polarities()	Infer the polarities of each LF based on evidence in a label matrix.
lf_summary([Y, est_weights])	Create a pandas DataFrame with the various per-LF statistics.

label_conflict()

Compute the fraction of data points with conflicting (non-abstain) labels.

Returns Fraction of data points with conflicting labels

Return type float

Example

```
>>> L = np.array([
...     [-1, 0, 0],
...     [-1, -1, -1],
...     [1, 0, -1],
...     [-1, 0, -1],
...     [0, 0, 0],
... ])
>>> LFAnalysis(L).label_conflict()
0.2
```

label_coverage()

Compute the fraction of data points with at least one label.

Returns Fraction of data points with labels

Return type float

Example

```
>>> L = np.array([
...     [-1, 0, 0],
...     [-1, -1, -1],
...     [1, 0, -1],
...     [-1, 0, -1],
...     [0, 0, 0],
... ])
>>> LFAnalysis(L).label_coverage()
0.8
```

label_overlap()

Compute the fraction of data points with at least two (non-abstain) labels.

Returns Fraction of data points with overlapping labels

Return type float

Example

```
>>> L = np.array([
...     [-1, 0, 0],
...     [-1, -1, -1],
...     [1, 0, -1],
...     [-1, 0, -1],
...     [0, 0, 0],
... ])
>>> LFAnalysis(L).label_overlap()
0.6
```

lf_conflicts(*normalize_by_overlaps=False*)

Compute frac. of examples each LF labels and labeled differently by another LF.

A conflicting example is one that at least one other LF returns a different (non-abstain) label for.

Note that the maximum possible conflict fraction for an LF is the LF's overlaps fraction, unless `normalize_by_overlaps=True`, in which case it is 1.

Parameters **normalize_by_overlaps** (`bool`) – Normalize by overlaps of the LF, so that it returns the percent of LF overlaps that have conflicts.

Returns Fraction of conflicting examples for each LF

Return type numpy.ndarray

Example

```
>>> L = np.array([
...     [-1, 0, 0],
...     [-1, -1, -1],
...     [1, 0, -1],
...     [-1, 0, -1],
...     [0, 0, 0],
... ])
>>> LFAnalysis(L).lf_conflicts()
array([0.2, 0.2, 0. ])
>>> LFAnalysis(L).lf_conflicts(normalize_by_overlaps=True)
array([0.5       , 0.33333333, 0.        ])
```

lf_coverages()

Compute frac. of examples each LF labels.

Returns Fraction of labeled examples for each LF

Return type numpy.ndarray

Example

```
>>> L = np.array([
...     [-1, 0, 0],
...     [-1, -1, -1],
...     [1, 0, -1],
```

(continues on next page)

(continued from previous page)

```
...      [-1, 0, -1],
...      [0, 0, 0],
... ])
>>> LFAnalysis(L).lf_coverages()
array([0.4, 0.8, 0.4])
```

lf_empirical_accuracies(*Y*)

Compute empirical accuracy against a set of labels Y for each LF.

Usually, Y represents development set labels.

Parameters **Y** (ndarray) – [n] or [n, 1] np.ndarray of gold labels

Returns Empirical accuracies for each LF

Return type numpy.ndarray

lf_empirical_probs(*Y*, *k*)

Estimate conditional probability tables for each LF.

Computes conditional probability tables, P(L | Y), for each LF using the provided true labels Y.

Parameters

- **Y** (ndarray) – The n-dim array of true labels in {1,. . . ,k}
- **k** (int) – The cardinality i.e. number of classes

Returns An m x (k+1) x k np.ndarray representing the m (k+1) x k conditional probability tables P_i, where P_i[l,y] represents P(LF_i = l | Y = y) empirically calculated

Return type np.ndarray

lf_overlaps(*normalize_by_coverage=False*)

Compute frac. of examples each LF labels that are labeled by another LF.

An overlapping example is one that at least one other LF returns a (non-abstain) label for.

Note that the maximum possible overlap fraction for an LF is the LF's coverage, unless `normalize_by_coverage=True`, in which case it is 1.

Parameters **normalize_by_coverage** (bool) – Normalize by coverage of the LF, so that it returns the percent of LF labels that have overlaps.

Returns Fraction of overlapping examples for each LF

Return type numpy.ndarray

Example

```
>>> L = np.array([
...      [-1, 0, 0],
...      [-1, -1, -1],
...      [1, 0, -1],
...      [-1, 0, -1],
...      [0, 0, 0],
... ])
>>> LFAnalysis(L).lf_overlaps()
array([0.4, 0.6, 0.4])
>>> LFAnalysis(L).lf_overlaps(normalize_by_coverage=True)
array([1.  , 0.75, 1.  ])
```

lf_polarities()
Infer the polarities of each LF based on evidence in a label matrix.

Returns Unique output labels for each LF

Return type List[List[int]]

Example

```
>>> L = np.array([
...     [-1, 0, 0],
...     [-1, -1, -1],
...     [1, 0, -1],
...     [-1, 0, -1],
...     [0, 0, 0],
... ])
>>> LFAnalysis(L).lf_polarities()
[[0, 1], [0], [0]]
```

lf_summary(*Y=None*, *est_weights=None*)
Create a pandas DataFrame with the various per-LF statistics.

Parameters

- **Y** (Optional[ndarray]) – [n] or [n, 1] np.ndarray of gold labels. If provided, the empirical weight for each LF will be calculated.
- **est_weights** (Optional[ndarray]) – Learned weights for each LF

Returns Summary statistics for each LF

Return type pandas.DataFrame

4.3 snorkel.labeling.LFApplier

class snorkel.labeling.**LFApplier**(*lfs*)
Bases: snorkel.labeling.apply.core.BaseLFApplier

LF applier for a list of data points (e.g. SimpleNamespace) or a NumPy array.

Parameters **lfs** (List[LabelingFunction]) – LFs that this applier executes on examples

Example

```
>>> from snorkel.labeling import labeling_function
>>> @labeling_function()
... def is_big_num(x):
...     return 1 if x.num > 42 else 0
>>> applier = LFApplier([is_big_num])
>>> from types import SimpleNamespace
>>> applier.apply([SimpleNamespace(num=10), SimpleNamespace(num=100)])
array([[0], [1]])
```

```
>>> @labeling_function()
... def is_big_num_np(x):
...     return 1 if x[0] > 42 else 0
>>> applier = LFApplier([is_big_num_np])
>>> applier.apply(np.array([[10], [100]]))
array([[0], [1]])
```

__init__ (*lfs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(data_points[, progress_bar, ...])	Label list of data points or a NumPy array with LFs.

apply (*data_points*, *progress_bar=True*, *fault_tolerant=False*, *return_meta=False*)
Label list of data points or a NumPy array with LFs.

Parameters

- **data_points** (Union[Sequence[Any], ndarray]) – List of data points or NumPy array to be labeled by LFs
- **progress_bar** (bool) – Display a progress bar?
- **fault_tolerant** (bool) – Output -1 if LF execution fails?
- **return_meta** (bool) – Return metadata from apply call?

Return type Union[ndarray, Tuple[ndarray, ApplierMetadata]]

Returns

- *np.ndarray* – Matrix of labels emitted by LFs
- *ApplierMetadata* – Metadata, such as fault counts, for the apply call

4.4 snorkel.labeling.model.label_model.LabelModel

class snorkel.labeling.model.label_model.**LabelModel** (*cardinality=2*, ***kwargs*)
Bases: torch.nn.modules.module.Module, snorkel.labeling.model.base_labeler.BaseLabeler

A model for learning the LF accuracies and combining their output labels.

This class learns a model of the labeling functions' conditional probabilities of outputting the true (unobserved) label *Y*, *P(lf | Y)*, and uses this learned model to re-weight and combine their output labels.

This class is based on the approach in [Training Complex Models with Multi-Task Weak Supervision](https://arxiv.org/abs/1810.02840), published in AAAI'19. In this approach, we compute the inverse generalized covariance matrix of the junction tree of a given LF dependency graph, and perform a matrix completion-style approach with respect to these empirical statistics. The result is an estimate of the conditional LF probabilities, *P(lf | Y)*, which are then set as the parameters of the label model used to re-weight and combine the labels output by the LFs.

Currently this class uses a conditionally independent label model, in which the LFs are assumed to be conditionally independent given *Y*.

Examples

```
>>> label_model = LabelModel()
>>> label_model = LabelModel(cardinality=3)
>>> label_model = LabelModel(cardinality=3, device='cpu')
>>> label_model = LabelModel(cardinality=3)
```

Parameters

- **cardinality** (int) – Number of classes, by default 2
- ****kwargs** – Arguments for changing config defaults

Raises **ValueError** – If config device set to cuda but only cpu is available

cardinality
Number of classes, by default 2

config
Training configuration

seed
Random seed

__init__(*cardinality=2*, ***kwargs*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

Return type None

Methods

__init__([cardinality])	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
fit(L_train[, Y_dev, class_balance])	Train label model.
float()	Casts all floating point parameters and buffers to float datatype.
forward(*input)	Defines the computation performed at every call.

Continued on next page

Table 5 – continued from previous page

get_conditional_probs()	Return the estimated conditional probabilities table.
get_weights()	Return the vector of learned LF weights for combining LFs.
half()	Casts all floating point parameters and buffers to half datatype.
load(source)	Load existing label model.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
predict(L[, return_probs, tie_break_policy])	Return predicted labels, with ties broken according to policy.
predict_proba(L)	Return label probabilities P(Y | lambda).
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
save(destination)	Save label model.
score(L, Y[, metrics, tie_break_policy])	Calculate one or more scores from user-specified and/or user-defined metrics.
share_memory()	**rtype** ~T
state_dict([destination, prefix, keep_vars])	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination	

Continued on next page

Table 6 – continued from previous page

dump_patches

fit (*L_train*, *Y_dev=None*, *class_balance=None*, ***kwargs*)

Train label model.

Train label model to estimate mu, the parameters used to combine LFs.

Parameters

- **L_train** (ndarray) – An [n,m] matrix with values in {-1,0,1,. . . ,k-1}
- **Y_dev** (Optional[ndarray]) – Gold labels for dev set for estimating class_balance, by default None
- **class_balance** (Optional[List[float]]) – Each class's percentage of the population, by default None
- ****kwargs** – Arguments for changing train config defaults.

  n_epochs The number of epochs to train (where each epoch is a single optimization step), default is 100

  lr Base learning rate (will also be affected by lr_scheduler choice and settings), default is 0.01

  l2 Centered L2 regularization strength, default is 0.0

  optimizer Which optimizer to use (one of ["sgd", "adam", "adamax"]), default is "sgd"

  optimizer_config Settings for the optimizer

  lr_scheduler Which lr_scheduler to use (one of ["constant", "linear", "exponential", "step"]), default is "constant"

  lr_scheduler_config Settings for the LRScheduler

  prec_init LF precision initializations / priors, default is 0.7

  seed A random seed to initialize the random number generator with

  log_freq Report loss every this many epochs (steps), default is 10

  mu_eps Restrict the learned conditional probabilities to [mu_eps, 1-mu_eps], default is None

Raises **Exception** – If loss in NaN

Examples

```
>>> L = np.array([[0, 0, -1], [-1, 0, 1], [1, -1, 0]])
>>> Y_dev = [0, 1, 0]
>>> label_model = LabelModel(verbose=False)
>>> label_model.fit(L)
>>> label_model.fit(L, Y_dev=Y_dev, seed=2020, lr=0.05)
>>> label_model.fit(L, class_balance=[0.7, 0.3], n_epochs=200, l2=0.4)
```

Return type None

forward(**input*)
Defines the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

Return type `None`

get_conditional_probs()
Return the estimated conditional probabilities table.

Return the estimated conditional probabilites table cprobs, where cprobs is an (m, k+1, k)-dim np.ndarray with:

cprobs[i, j, k] = P(lf_i = j-1 | Y = k)

where m is the number of LFs, k is the cardinality, and cprobs includes the conditional abstain probabilities P(lf_i = -1 | Y = y).

Returns An [m, k + 1, k] np.ndarray conditional probabilities table.

Return type np.ndarray

get_weights()
Return the vector of learned LF weights for combining LFs.

Returns [m,1] vector of learned LF weights for combining LFs.

Return type np.ndarray

Example

```
>>> L = np.array([[1, 1, 1], [1, 1, -1], [-1, 0, 0], [0, 0, 0]])
>>> label_model = LabelModel(verbose=False)
>>> label_model.fit(L, seed=123)
>>> np.around(label_model.get_weights(), 2)  # doctest: +SKIP
array([0.99, 0.99, 0.99])
```

load(*source*)
Load existing label model.

Parameters **source** (`str`) – Filename to load model from

Example

Load parameters saved in `saved_label_model`

```
>>> label_model.load('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type `None`

predict (*L*, *return_probs=False*, *tie_break_policy='abstain'*)
Return predicted labels, with ties broken according to policy.

Policies to break ties include:

- "abstain": return an abstain vote (-1)
- "true-random": randomly choose among the tied options
- "random": randomly choose among tied option using deterministic hash

NOTE: if tie_break_policy="true-random", repeated runs may have slightly different results due to difference in broken ties

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,...,k-1}
- **return_probs** (Optional[bool]) – Whether to return probs along with preds
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Return type Union[ndarray, Tuple[ndarray, ndarray]]

Returns

- *np.ndarray* – An [n,1] array of integer labels
- *(np.ndarray, np.ndarray)* – An [n,1] array of integer labels and an [n,k] array of probabilistic labels

Example

```
>>> L = np.array([[0, 0, -1], [1, 1, -1], [0, 0, -1]])
>>> label_model = LabelModel(verbose=False)
>>> label_model.fit(L)
>>> label_model.predict(L)
array([0, 1, 0])
```

predict_proba (*L*)
Return label probabilities P(Y | lambda).

Parameters **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,...,k-1}f

Returns An [n,k] array of probabilistic labels

Return type np.ndarray

Example

```
>>> L = np.array([[0, 0, 0], [1, 1, 1], [1, 1, 1]])
>>> label_model = LabelModel(verbose=False)
>>> label_model.fit(L, seed=123)
>>> np.around(label_model.predict_proba(L), 1)  # doctest: +SKIP
array([[1., 0.],
       [0., 1.],
       [0., 1.]])
```

save (*destination*)
Save label model.

Parameters **destination** (str) – Filename for saving model

Example

```
>>> label_model.save('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

score(*L, Y, metrics=['accuracy'], tie_break_policy='abstain'*)

Calculate one or more scores from user-specified and/or user-defined metrics.

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,. . . ,k-1}
- **Y** (ndarray) – Gold labels associated with data points in L
- **metrics** (Optional[List[str]]) – A list of metric names. Possbile metrics are - *accuracy*, *coverage*, *precision*, *recall*, *f1*, *f1_micro*, *f1_macro*, *fbeta*, *matthews_corrcoef*, *roc_auc*. See sklearn.metrics for details on the metrics.
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions. Same as predict() method above.

Returns A dictionary mapping metric names to metric scores

Return type Dict[str, float]

Example

```
>>> L = np.array([[1, 1, -1], [0, 0, -1], [1, 1, -1]])
>>> label_model = LabelModel(verbose=False)
>>> label_model.fit(L)
>>> label_model.score(L, Y=np.array([1, 1, 1]))
{'accuracy': 0.6666666666666666}
>>> label_model.score(L, Y=np.array([1, 1, 1]), metrics=["f1"])
{'f1': 0.8}
```

4.5 snorkel.labeling.LabelingFunction

class snorkel.labeling.**LabelingFunction**(*name*, *f*, *resources=None*, *pre=None*)

Bases: object

Base class for labeling functions.

A labeling function (LF) is a function that takes a data point as input and produces an integer label, corresponding to a class. A labeling function can also abstain from voting by outputting -1. For examples, see the Snorkel tutorials.

This class wraps a Python function outputting a label. Extra functionality, such as running preprocessors and storing resources, is provided. Simple LFs can be defined via a decorator. See labeling_function.

Parameters

- **name** (str) – Name of the LF

- **f** (Callable[..., int]) – Function that implements the core LF logic
- **resources** (Optional[Mapping[str, Any]]) – Labeling resources passed in to f via kwargs
- **pre** (Optional[List[BaseMapper]]) – Preprocessors to run on data points before LF execution

Raises **ValueError** – Calling incorrectly defined preprocessors

name
See above

__init__(*name*, *f*, *resources=None*, *pre=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, f[, resources, pre])	Initialize self.

__call__(*x*)
Label data point.

Runs all preprocessors, then passes preprocessed data point to LF.

Parameters **x** (Any) – Data point to label

Returns Label for data point

Return type int

4.6 snorkel.labeling.model.baselines.MajorityClassVoter

class snorkel.labeling.model.baselines.**MajorityClassVoter**(*cardinality=2*, ***kwargs*)

Bases: snorkel.labeling.model.base_labeler.BaseLabeler

Majority class label model.

__init__(*cardinality=2*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([cardinality])	Initialize self.
fit(balance, *args, **kwargs)	Train majority class model.
load(source)	Load existing label model.
predict(L[, return_probs, tie_break_policy])	Return predicted labels, with ties broken according to policy.
predict_proba(L)	Predict probabilities using majority class.
save(destination)	Save label model.

Continued on next page

Table 8 – continued from previous page

score(L, Y[, metrics, tie_break_policy])	Calculate one or more scores from user-specified and/or user-defined metrics.

fit (*balance*, **args*, ***kwargs*)
Train majority class model.

Set class balance for majority class label model.

Parameters **balance** (ndarray) – A [k] array of class probabilities

Return type None

load (*source*)
Load existing label model.

Parameters **source** (str) – Filename to load model from

Example

Load parameters saved in saved_label_model

```
>>> label_model.load('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

predict (*L*, *return_probs=False*, *tie_break_policy='abstain'*)
Return predicted labels, with ties broken according to policy.

Policies to break ties include: "abstain": return an abstain vote (-1) "true-random": randomly choose among the tied options "random": randomly choose among tied option using deterministic hash

NOTE: if tie_break_policy="true-random", repeated runs may have slightly different results due to difference in broken ties

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,. . . ,k-1}
- **return_probs** (Optional[bool]) – Whether to return probs along with preds
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Return type Union[ndarray, Tuple[ndarray, ndarray]]

Returns

- *np.ndarray* – An [n,1] array of integer labels
- *(np.ndarray, np.ndarray)* – An [n,1] array of integer labels and an [n,k] array of probabilistic labels

predict_proba (*L*)
Predict probabilities using majority class.

Assign majority class vote to each datapoint. In case of multiple majority classes, assign equal probabilities among them.

Parameters **L** (ndarray) – An [n, m] matrix of labels

Returns A [n, k] array of probabilistic labels

Return type np.ndarray

Example

```
>>> L = np.array([[0, 0, -1], [-1, 0, 1], [1, -1, 0]])
>>> maj_class_voter = MajorityClassVoter()
>>> maj_class_voter.fit(balance=np.array([0.8, 0.2]))
>>> maj_class_voter.predict_proba(L)
array([[1., 0.],
       [1., 0.],
       [1., 0.]])
```

save(*destination*)
Save label model.

Parameters **destination** (str) – Filename for saving model

Example

```
>>> label_model.save('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

score(*L, Y, metrics=['accuracy'], tie_break_policy='abstain'*)
Calculate one or more scores from user-specified and/or user-defined metrics.

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,. . . ,k-1}
- **Y** (ndarray) – Gold labels associated with data points in L
- **metrics** (Optional[List[str]]) – A list of metric names
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Returns A dictionary mapping metric names to metric scores

Return type Dict[str, float]

4.7 snorkel.labeling.model.baselines.MajorityLabelVoter

class snorkel.labeling.model.baselines.**MajorityLabelVoter**(*cardinality=2*, ***kwargs*)

Bases: snorkel.labeling.model.base_labeler.BaseLabeler

Majority vote label model.

__init__(*cardinality=2*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([cardinality])	Initialize self.
load(source)	Load existing label model.
predict(L[, return_probs, tie_break_policy])	Return predicted labels, with ties broken according to policy.
predict_proba(L)	Predict probabilities using majority vote.
save(destination)	Save label model.
score(L, Y[, metrics, tie_break_policy])	Calculate one or more scores from user-specified and/or user-defined metrics.

load(*source*)
Load existing label model.

Parameters **source** (str) – Filename to load model from

Example

Load parameters saved in `saved_label_model`

```
>>> label_model.load('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

predict(*L*, *return_probs=False*, *tie_break_policy='abstain'*)
Return predicted labels, with ties broken according to policy.

Policies to break ties include: "abstain": return an abstain vote (-1) "true-random": randomly choose among the tied options "random": randomly choose among tied option using deterministic hash

NOTE: if tie_break_policy="true-random", repeated runs may have slightly different results due to difference in broken ties

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,. . . ,k-1}
- **return_probs** (Optional[bool]) – Whether to return probs along with preds
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Return type Union[ndarray, Tuple[ndarray, ndarray]]

Returns

- *np.ndarray* – An [n,1] array of integer labels
- *(np.ndarray, np.ndarray)* – An [n,1] array of integer labels and an [n,k] array of probabilistic labels

predict_proba(*L*)
Predict probabilities using majority vote.

Assign vote by calculating majority vote across all labeling functions. In case of ties, non-integer probabilities are possible.

Parameters **L** (ndarray) – An [n, m] matrix of labels

Returns A [n, k] array of probabilistic labels

Return type np.ndarray

Example

```
>>> L = np.array([[0, 0, -1], [-1, 0, 1], [1, -1, 0]])
>>> maj_voter = MajorityLabelVoter()
>>> maj_voter.predict_proba(L)
array([[1. , 0. ],
       [0.5, 0.5],
       [0.5, 0.5]])
```

save(*destination*)

Save label model.

Parameters **destination** (str) – Filename for saving model

Example

```
>>> label_model.save('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

score(*L, Y, metrics=['accuracy'], tie_break_policy='abstain'*)

Calculate one or more scores from user-specified and/or user-defined metrics.

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,. . . ,k-1}
- **Y** (ndarray) – Gold labels associated with data points in L
- **metrics** (Optional[List[str]]) – A list of metric names
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Returns A dictionary mapping metric names to metric scores

Return type Dict[str, float]

4.8 snorkel.labeling.lf.nlp.NLPLabelingFunction

class snorkel.labeling.lf.nlp.**NLPLabelingFunction**(*name*, *f*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Bases: snorkel.labeling.lf.nlp.BaseNLPLabelingFunction

Special labeling function type for spaCy-based LFs.

This class is a special version of LabelingFunction. It has a SpacyPreprocessor integrated which shares a cache with all other NLPLabelingFunction instances. This makes it easy to define

LFs that have a text input field and have logic written over spaCy `Doc` objects. Examples passed into an `NLPLabelingFunction` will have a new field which can be accessed which contains a spaCy `Doc`. By default, this field is called `doc`. A `Doc` object is a sequence of `Token` objects, which contain information on lemmatization, parts-of-speech, etc. `Doc` objects also contain fields like `Doc.ents`, a list of named entities, and `Doc.noun_chunks`, a list of noun phrases. For details of spaCy `Doc` objects and a full attribute listing, see https://spacy.io/api/doc.

Simple `NLPLabelingFunctions` can be defined via a decorator. See `nlp_labeling_function`.

Parameters

- **name** (`str`) – Name of the LF
- **f** (`Callable[..., int]`) – Function that implements the core LF logic
- **resources** (`Optional[Mapping[str, Any]]`) – Labeling resources passed in to `f` via `kwargs`
- **pre** (`Optional[List[BaseMapper]]`) – Preprocessors to run before SpacyPreprocessor is executed
- **text_field** (`str`) – Name of data point text field to input
- **doc_field** (`str`) – Name of data point field to output parsed document to
- **language** (`str`) – spaCy model to load See https://spacy.io/usage/models#usage
- **disable** (`Optional[List[str]]`) – List of pipeline components to disable See https://spacy.io/usage/processing-pipelines#disabling
- **memoize** (`bool`) – Memoize preprocessor outputs?
- **memoize_key** (`Optional[Callable[[Any], Hashable]]`) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)
- **gpu** (`bool`) – Prefer Spacy GPU processing?

Raises **ValueError** – Calling incorrectly defined preprocessors

Example

```
>>> def f(x):
...     person_ents = [ent for ent in x.doc.ents if ent.label_ == "PERSON"]
...     return 0 if len(person_ents) > 0 else -1
>>> has_person_mention = NLPLabelingFunction(name="has_person_mention", f=f)
>>> has_person_mention
NLPLabelingFunction has_person_mention, Preprocessors: [SpacyPreprocessor...]
```

```
>>> from types import SimpleNamespace
>>> x = SimpleNamespace(text="The movie was good.")
>>> has_person_mention(x)
-1
```

name
See above

__init__(*name*, *f*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)
Initialize self. See help(type(self)) for accurate signature.

Return type `None`

Methods

__init__(name, f[, resources, pre, ...])	Initialize self.

__call__ (*x*)
Label data point.

Runs all preprocessors, then passes preprocessed data point to LF.

Parameters **x** (Any) – Data point to label

Returns Label for data point

Return type int

4.9 snorkel.labeling.PandasLFApplier

class snorkel.labeling.**PandasLFApplier**(*lfs*)
Bases: snorkel.labeling.apply.core.BaseLFApplier

LF applier for a Pandas DataFrame.

Data points are stored as Series in a DataFrame. The LFs are executed via a pandas.DataFrame.apply call, which is single-process and can be slow for large DataFrames. For large datasets, consider DaskLFApplier or SparkLFApplier.

Parameters **lfs** (List[LabelingFunction]) – LFs that this applier executes on examples

Example

```
>>> from snorkel.labeling import labeling_function
>>> @labeling_function()
... def is_big_num(x):
...     return 1 if x.num > 42 else 0
>>> applier = PandasLFApplier([is_big_num])
>>> applier.apply(pd.DataFrame(dict(num=[10, 100], text=["hello", "hi"])))
array([[0], [1]])
```

__init__ (*lfs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(df[, progress_bar, fault_tolerant, ...])	Label Pandas DataFrame of data points with LFs.

apply (*df*, *progress_bar=True*, *fault_tolerant=False*, *return_meta=False*)
Label Pandas DataFrame of data points with LFs.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data points to be labeled by LFs

- **progress_bar** (bool) – Display a progress bar?
- **fault_tolerant** (bool) – Output −1 if LF execution fails?
- **return_meta** (bool) – Return metadata from apply call?

Return type Union[ndarray, Tuple[ndarray, ApplierMetadata]]

Returns

- *np.ndarray* – Matrix of labels emitted by LFs
- *ApplierMetadata* – Metadata, such as fault counts, for the apply call

4.10 snorkel.labeling.apply.dask.PandasParallelLFApplier

class snorkel.labeling.apply.dask.**PandasParallelLFApplier**(*lfs*)

Bases: *snorkel.labeling.apply.dask.DaskLFApplier*

Parallel LF applier for a Pandas DataFrame.

Creates a Dask DataFrame from a Pandas DataFrame, then uses DaskLFApplier to label data in parallel. See DaskLFApplier.

__init__(*lfs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(df[, n_parallel, scheduler, ...])	Label Pandas DataFrame of data points with LFs in parallel using Dask.

apply(*df*, *n_parallel=2*, *scheduler='processes'*, *fault_tolerant=False*)

Label Pandas DataFrame of data points with LFs in parallel using Dask.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data points to be labeled by LFs
- **n_parallel** (int) – Parallelism level for LF application. Corresponds to npartitions in constructed Dask DataFrame. For scheduler="processes", number of processes launched. Recommended to be no more than the number of cores on the running machine.
- **scheduler** (Union[str, dask.distributed.Client]) – A Dask scheduling configuration: either a string option or a Client. For more information, see https://docs.dask.org/en/stable/scheduling.html#
- **fault_tolerant** (bool) – Output −1 if LF execution fails?

Returns Matrix of labels emitted by LFs

Return type np.ndarray

4.11 snorkel.labeling.model.baselines.RandomVoter

class snorkel.labeling.model.baselines.**RandomVoter**(*cardinality=2*, ***kwargs*)

Bases: snorkel.labeling.model.base_labeler.BaseLabeler

Random vote label model.

Example

```
>>> L = np.array([[0, 0, -1], [-1, 0, 1], [1, -1, 0]])
>>> random_voter = RandomVoter()
>>> predictions = random_voter.predict_proba(L)
```

__init__(*cardinality=2*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([cardinality])	Initialize self.
load(source)	Load existing label model.
predict(L[, return_probs, tie_break_policy])	Return predicted labels, with ties broken according to policy.
predict_proba(L)	Assign random votes to the data points.
save(destination)	Save label model.
score(L, Y[, metrics, tie_break_policy])	Calculate one or more scores from user-specified and/or user-defined metrics.

load(*source*)

Load existing label model.

Parameters **source** (str) – Filename to load model from

Example

Load parameters saved in saved_label_model

```
>>> label_model.load('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

predict(*L*, *return_probs=False*, *tie_break_policy='abstain'*)

Return predicted labels, with ties broken according to policy.

Policies to break ties include: "abstain": return an abstain vote (-1) "true-random": randomly choose among the tied options "random": randomly choose among tied option using deterministic hash

NOTE: if tie_break_policy="true-random", repeated runs may have slightly different results due to difference in broken ties

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,...,k-1}
- **return_probs** (Optional[bool]) – Whether to return probs along with preds
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Return type Union[ndarray, Tuple[ndarray, ndarray]]

Returns

- *np.ndarray* – An [n,1] array of integer labels
- *(np.ndarray, np.ndarray)* – An [n,1] array of integer labels and an [n,k] array of probabilistic labels

predict_proba(*L*)

Assign random votes to the data points.

Parameters **L** (ndarray) – An [n, m] matrix of labels

Returns A [n, k] array of probabilistic labels

Return type np.ndarray

Example

```
>>> L = np.array([[0, 0, -1], [-1, 0, 1], [1, -1, 0]])
>>> random_voter = RandomVoter()
>>> predictions = random_voter.predict_proba(L)
```

save(*destination*)

Save label model.

Parameters **destination** (str) – Filename for saving model

Example

```
>>> label_model.save('./saved_label_model.pkl')  # doctest: +SKIP
```

Return type None

score(*L, Y, metrics=['accuracy'], tie_break_policy='abstain'*)

Calculate one or more scores from user-specified and/or user-defined metrics.

Parameters

- **L** (ndarray) – An [n,m] matrix with values in {-1,0,1,...,k-1}
- **Y** (ndarray) – Gold labels associated with data points in L
- **metrics** (Optional[List[str]]) – A list of metric names
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions

Returns A dictionary mapping metric names to metric scores

Return type Dict[str, float]

4.12 snorkel.labeling.apply.spark.SparkLFApplier

class snorkel.labeling.apply.spark.**SparkLFApplier**(*lfs*)

Bases: snorkel.labeling.apply.core.BaseLFApplier

LF applier for a Spark RDD.

Data points are stored as Rows in an RDD, and a Spark map job is submitted to execute the LFs. A common way to obtain an RDD is via a PySpark DataFrame. For an example usage with AWS EMR instructions, see test/labeling/apply/lf_applier_spark_test_script.py.

__init__(*lfs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(data_points[, fault_tolerant])	Label PySpark RDD of data points with LFs.

apply(*data_points*, *fault_tolerant=False*)

Label PySpark RDD of data points with LFs.

Parameters

- **data_points** (*pyspark.RDD*) – PySpark RDD containing data points to be labeled by LFs
- **fault_tolerant** (bool) – Output -1 if LF execution fails?

Returns Matrix of labels emitted by LFs

Return type np.ndarray

4.13 snorkel.labeling.lf.nlp_spark.SparkNLPLabelingFunction

class snorkel.labeling.lf.nlp_spark.**SparkNLPLabelingFunction**(*name*, *f*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Bases: snorkel.labeling.lf.nlp.BaseNLPLabelingFunction

Special labeling function type for SpaCy-based LFs running on Spark.

This class is a Spark-compatible version of NLPLabelingFunction. See NLPLabelingFunction for details.

Parameters

- **name** (str) – Name of the LF
- **f** (Callable[..., int]) – Function that implements the core LF logic
- **resources** (Optional[Mapping[str, Any]]) – Labeling resources passed in to f via kwargs
- **pre** (Optional[List[BaseMapper]]) – Preprocessors to run before SpacyPreprocessor is executed
- **text_field** (str) – Name of data point text field to input
- **doc_field** (str) – Name of data point field to output parsed document to
- **language** (str) – SpaCy model to load See https://spacy.io/usage/models#usage
- **disable** (Optional[List[str]]) – List of pipeline components to disable See https://spacy.io/usage/processing-pipelines#disabling
- **memoize** (bool) – Memoize preprocessor outputs?
- **memoize_key** (Optional[Callable[[Any], Hashable]]) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)
- **gpu** (bool) – Prefer Spacy GPU processing?

Raises **ValueError** – Calling incorrectly defined preprocessors

name
See above

__init__(*name*, *f*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, f[, resources, pre, ...])	Initialize self.

__call__(*x*)
Label data point.

Runs all preprocessors, then passes preprocessed data point to LF.

Parameters **x** (Any) – Data point to label

Returns Label for data point

Return type int

4.14 snorkel.labeling.filter_unlabeled_dataframe

snorkel.labeling.**filter_unlabeled_dataframe**(*X*, *y*, *L*)
Filter out examples not covered by any labeling function.

Parameters

- **X** (DataFrame) – Data points in a Pandas DataFrame.
- **y** (ndarray) – Matrix of probabilities output by label model's predict_proba method.
- **L** (ndarray) – Matrix of labels emitted by LFs.

Return type Tuple[DataFrame, ndarray]

Returns

- *pd.DataFrame* – Data points that were labeled by at least one LF in L.
- *np.ndarray* – Probabilities matrix for data points labeled by at least one LF in L.

4.15 snorkel.labeling.labeling_function

class snorkel.labeling.**labeling_function**(*name=None*, *resources=None*, *pre=None*)

Bases: object

Decorator to define a LabelingFunction object from a function.

Parameters

- **name** (Optional[str]) – Name of the LF
- **resources** (Optional[Mapping[str, Any]]) – Labeling resources passed in to f via kwargs
- **pre** (Optional[List[BaseMapper]]) – Preprocessors to run on data points before LF execution

Examples

```
>>> @labeling_function()
... def f(x):
...     return 0 if x.a > 42 else -1
>>> f
LabelingFunction f, Preprocessors: []
>>> from types import SimpleNamespace
>>> x = SimpleNamespace(a=90, b=12)
>>> f(x)
0
```

```
>>> @labeling_function(name="my_lf")
... def g(x):
...     return 0 if x.a > 42 else -1
>>> g
LabelingFunction my_lf, Preprocessors: []
```

__init__(*name=None*, *resources=None*, *pre=None*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, resources, pre])	Initialize self.

__call__(*f*)

Wrap a function to create a `LabelingFunction`.

Parameters **f** (`Callable[..., int]`) – Function that implements the core LF logic

Returns New `LabelingFunction` executing logic in wrapped function

Return type *LabelingFunction*

4.16 snorkel.labeling.lf.nlp.nlp_labeling_function

class snorkel.labeling.lf.nlp.**nlp_labeling_function**(*name=None*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Bases: `snorkel.labeling.lf.nlp.base_nlp_labeling_function`

Decorator to define an NLPLabelingFunction object from a function.

Parameters

- **name** (`Optional[str]`) – Name of the LF
- **resources** (`Optional[Mapping[str, Any]]`) – Labeling resources passed in to `f` via `kwargs`
- **pre** (`Optional[List[BaseMapper]]`) – Preprocessors to run before SpacyPreprocessor is executed
- **text_field** (`str`) – Name of data point text field to input
- **doc_field** (`str`) – Name of data point field to output parsed document to
- **language** (`str`) – spaCy model to load See https://spacy.io/usage/models#usage
- **disable** (`Optional[List[str]]`) – List of pipeline components to disable See https://spacy.io/usage/processing-pipelines#disabling
- **memoize** (`bool`) – Memoize preprocessor outputs?
- **memoize_key** (`Optional[Callable[[Any], Hashable]]`) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

Example

```
>>> @nlp_labeling_function()
... def has_person_mention(x):
...     person_ents = [ent for ent in x.doc.ents if ent.label_ == "PERSON"]
...     return 0 if len(person_ents) > 0 else -1
>>> has_person_mention
NLPLabelingFunction has_person_mention, Preprocessors: [SpacyPreprocessor...]
```

```
>>> from types import SimpleNamespace
>>> x = SimpleNamespace(text="The movie was good.")
>>> has_person_mention(x)
-1
```

__init__(*name=None*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, resources, pre, text_field, ...])	Initialize self.

__call__(*f*)

Wrap a function to create an BaseNLPLabelingFunction.

Parameters **f** (Callable[..., int]) – Function that implements the core NLP LF logic

Returns New BaseNLPLabelingFunction executing logic in wrapped function

Return type BaseNLPLabelingFunction

4.17 snorkel.labeling.lf.nlp_spark.spark_nlp_labeling_function

class snorkel.labeling.lf.nlp_spark.**spark_nlp_labeling_function**(*name=None*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Bases: snorkel.labeling.lf.nlp.base_nlp_labeling_function

Decorator to define a SparkNLPLabelingFunction object from a function.

Parameters

- **name** (Optional[str]) – Name of the LF
- **resources** (Optional[Mapping[str, Any]]) – Labeling resources passed in to f via kwargs
- **pre** (Optional[List[BaseMapper]]) – Preprocessors to run before SpacyPreprocessor is executed
- **text_field** (str) – Name of data point text field to input

- **doc_field** (str) – Name of data point field to output parsed document to
- **language** (str) – SpaCy model to load See https://spacy.io/usage/models#usage
- **disable** (Optional[List[str]]) – List of pipeline components to disable See https://spacy.io/usage/processing-pipelines#disabling
- **memoize** (bool) – Memoize preprocessor outputs?
- **memoize_key** (Optional[Callable[[Any], Hashable]]) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

Example

```
>>> @spark_nlp_labeling_function()
... def has_person_mention(x):
...     person_ents = [ent for ent in x.doc.ents if ent.label_ == "PERSON"]
...     return 0 if len(person_ents) > 0 else -1
>>> has_person_mention
SparkNLPLabelingFunction has_person_mention, Preprocessors: [SpacyPreprocessor...]
```

```
>>> from pyspark.sql import Row
>>> x = Row(text="The movie was good.")
>>> has_person_mention(x)
-1
```

__init__(*name=None*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, resources, pre, text_field, ...])	Initialize self.

__call__(*f*)

Wrap a function to create an BaseNLPLabelingFunction.

Parameters **f** (Callable[..., int]) – Function that implements the core NLP LF logic

Returns New BaseNLPLabelingFunction executing logic in wrapped function

Return type BaseNLPLabelingFunction

CHAPTER

FIVE

SNORKEL MAP PACKAGE

Generic utilities for data point to data point operations.

BaseMapper	Base class for Mapper and LambdaMapper.
LambdaMapper	Define a mapper from a function.
Mapper	Base class for any data point to data point mapping in the pipeline.
lambda_mapper	Decorate a function to define a LambdaMapper object.
spark.make_spark_mapper	Convert Mapper to be compatible with PySpark.

5.1 snorkel.map.BaseMapper

class snorkel.map.**BaseMapper**(*name*, *pre*, *memoize*, *memoize_key=None*)

Bases: object

Base class for Mapper and LambdaMapper.

Implements nesting, memoization, and deep copy functionality. Used primarily for type checking.

Parameters

- **name** (str) – Name of the mapper
- **pre** (List[BaseMapper]) – Mappers to run before this mapper is executed
- **memoize** (bool) – Memoize mapper outputs?
- **memoize_key** (Optional[Callable[[Any], Hashable]]) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

Raises **NotImplementedError** – Subclasses need to implement _generate_mapped_data_point

memoize

Memoize mapper outputs?

__init__(*name*, *pre*, *memoize*, *memoize_key=None*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, pre, memoize[, memoize_key])	Initialize self.
reset_cache()	Reset the memoization cache.

__call__(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If `memoize` is set to `True`, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (`Any`) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type `None`

5.2 snorkel.map.LambdaMapper

class snorkel.map.**LambdaMapper**(*name*, *f*, *pre=None*, *memoize=False*, *memoize_key=None*)
Bases: `snorkel.map.core.BaseMapper`

Define a mapper from a function.

Convenience class for mappers that execute a simple function with no set up. The function should map from an input data point to a new data point directly, unlike `Mapper.run`. The original data point will not be updated, so in-place operations are safe.

Parameters

- **name** (`str`) – Name of mapper
- **f** (`Callable[[Any], Optional[Any]]`) – Function executing the mapping operation
- **pre** (`Optional[List[BaseMapper]]`) – Mappers to run before this mapper is executed
- **memoize** (`bool`) – Memoize mapper outputs?
- **memoize_key** (`Optional[Callable[[Any], Hashable]]`) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

__init__(*name*, *f*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type `None`

Methods

__init__(name, f[, pre, memoize, memoize_key])	Initialize self.
reset_cache()	Reset the memoization cache.

__call__(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If `memoize` is set to `True`, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (`Any`) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type `None`

5.3 snorkel.map.Mapper

class snorkel.map.**Mapper**(*name*, *field_names=None*, *mapped_field_names=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Bases: `snorkel.map.core.BaseMapper`

Base class for any data point to data point mapping in the pipeline.

Map data points to new data points by transforming, adding additional information, or decomposing into primitives. This module provides base classes for other operators like `TransformationFunction` and `Preprocessor`. We don't expect people to construct `Mapper` objects directly.

A Mapper maps an data point to a new data point, possibly with a different schema. Subclasses of Mapper need to implement the `run` method, which takes fields of the data point as input and outputs new fields for the mapped data point as a dictionary. The `run` method should only be called internally by the `Mapper` object, not directly by a user.

Mapper derivatives work for data points that have mutable attributes, like `SimpleNamespace`, `pd.Series`, or `dask.Series`. An example of a data point type without mutable fields is `pyspark.sql.Row`. Use `snorkel.map.spark.make_spark_mapper` for PySpark compatibility.

For an example of a Mapper, see `snorkel.preprocess.nlp.SpacyPreprocessor`

Parameters

- **name** (`str`) – Name of mapper
- **field_names** (`Optional[Mapping[str, str]]`) – A map from attribute names of the incoming data points to the input argument names of the `run` method. If None, the parameter names in the function signature are used.
- **mapped_field_names** (`Optional[Mapping[str, str]]`) – A map from output keys of the `run` method to attribute names of the output data points. If None, the original output keys are used.
- **pre** (`Optional[List[BaseMapper]]`) – Mappers to run before this mapper is executed
- **memoize** (`bool`) – Memoize mapper outputs?
- **memoize_key** (`Optional[Callable[[Any], Hashable]]`) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

Raises **NotImplementedError** – Subclasses must implement the `run` method

field_names
See above

mapped_field_names
See above

memoize
Memoize mapper outputs?

__init__(*name*, *field_names=None*, *mapped_field_names=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name[, field_names, ...])	Initialize self.
reset_cache()	Reset the memoization cache.
run(**kwargs)	Run the mapping operation using the input fields.

__call__(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If memoize is set to True, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (Any) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type None

run(***kwargs*)
Run the mapping operation using the input fields.

The inputs to this function are fed by extracting the fields of the input data point using the keys of field_names. The output field names are converted using mapped_field_names and added to the data point.

Returns A mapping from canonical output field names to their values.

Return type Optional[FieldMap]

Raises **NotImplementedError** – Subclasses must implement this method

5.4 snorkel.map.lambda_mapper

class snorkel.map.**lambda_mapper**(*name=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Bases: object

Decorate a function to define a LambdaMapper object.

Example

```
>>> @lambda_mapper()
... def concatenate_text(x):
...     x.article = f"{x.title} {x.body}"
...     return x
>>> isinstance(concatenate_text, LambdaMapper)
True
>>> from types import SimpleNamespace
>>> x = SimpleNamespace(title="my title", body="my text")
>>> concatenate_text(x).article
'my title my text'
```

Parameters

- **name** (Optional[str]) – Name of mapper. If None, uses the name of the wrapped function.
- **pre** (Optional[List[BaseMapper]]) – Mappers to run before this mapper is executed
- **memoize** (bool) – Memoize mapper outputs?
- **memoize_key** (Optional[Callable[[Any], Hashable]]) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

memoize
Memoize mapper outputs?

__init__(*name=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, pre, memoize, memoize_key])	Initialize self.

__call__(*f*)
Wrap a function to create a LambdaMapper.

Parameters **f** (Callable[[Any], Optional[Any]]) – Function executing the mapping operation

Returns New LambdaMapper executing operation in wrapped function

Return type *LambdaMapper*

5.5 snorkel.map.spark.make_spark_mapper

snorkel.map.spark.**make_spark_mapper**(*mapper*)
Convert Mapper to be compatible with PySpark.

Parameters **mapper** (Mapper) – Mapper to make compatible with PySpark

Return type Mapper

CHAPTER

SIX

SNORKEL PREPROCESS PACKAGE

Preprocessors for LFs, TFs, and SFs.

BasePreprocessor	alias of `snorkel.map.core.BaseMapper`
LambdaPreprocessor	Convenience class for defining preprocessors from functions.
Preprocessor	Base class for preprocessors.
nlp.SpacyPreprocessor	Preprocessor that parses input text via a SpaCy model.
spark.make_spark_preprocessor	Convert `Mapper` to be compatible with PySpark.
preprocessor	Decorate functions to create preprocessors.

6.1 snorkel.preprocess.BasePreprocessor

`snorkel.preprocess.`**`BasePreprocessor`**
alias of `snorkel.map.core.BaseMapper`

6.2 snorkel.preprocess.LambdaPreprocessor

class `snorkel.preprocess.`**`LambdaPreprocessor`**(*name*, *f*, *pre=None*, *memoize=False*, *memoize_key=None*)

Bases: `snorkel.map.core.LambdaMapper`

Convenience class for defining preprocessors from functions.

See `snorkel.map.core.LambdaMapper` for details.

`__init__`(*name*, *f*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type `None`

Methods

__init__(name, f[, pre, memoize, memoize_key])	Initialize self.
reset_cache()	Reset the memoization cache.

`__call__`(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If `memoize` is set to `True`, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (`Any`) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type `None`

6.3 snorkel.preprocess.Preprocessor

class snorkel.preprocess.**Preprocessor**(*name*, *field_names=None*, *mapped_field_names=None*, *pre=None*, *memoize=False*, *memoize_key=None*)

Bases: `snorkel.map.core.Mapper`

Base class for preprocessors.

See `snorkel.map.core.Mapper` for details.

__init__(*name*, *field_names=None*, *mapped_field_names=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type `None`

Methods

__init__(name[, field_names, ...])	Initialize self.
reset_cache()	Reset the memoization cache.
run(**kwargs)	Run the mapping operation using the input fields.

__call__(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If `memoize` is set to `True`, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (`Any`) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type `None`

run(***kwargs*)
Run the mapping operation using the input fields.

The inputs to this function are fed by extracting the fields of the input data point using the keys of `field_names`. The output field names are converted using `mapped_field_names` and added to the data point.

Returns A mapping from canonical output field names to their values.

Return type Optional[FieldMap]

Raises **`NotImplementedError`** – Subclasses must implement this method

6.4 snorkel.preprocess.nlp.SpacyPreprocessor

class snorkel.preprocess.nlp.**SpacyPreprocessor**(*text_field*, *doc_field*, *language='en_core_web_sm'*, *disable=None*, *pre=None*, *memoize=False*, *memoize_key=None*, *gpu=False*)

Bases: `snorkel.preprocess.core.Preprocessor`

Preprocessor that parses input text via a SpaCy model.

A common approach to writing LFs over text is to first use a natural language parser to decompose the text into tokens, part-of-speech tags, etc. SpaCy (https://spacy.io/) is a popular tool for doing this. This preprocessor adds a SpaCy `Doc` object to the data point. A `Doc` object is a sequence of `Token` objects, which contain information on lemmatization, parts-of-speech, etc. `Doc` objects also contain fields like `Doc.ents`, a list of named entities, and `Doc.noun_chunks`, a list of noun phrases. For details of SpaCy `Doc` objects and a full attribute listing, see https://spacy.io/api/doc.

Parameters

- **`text_field`** (`str`) – Name of data point text field to input
- **`doc_field`** (`str`) – Name of data point field to output parsed document to
- **`language`** (`str`) – SpaCy model to load See https://spacy.io/usage/models#usage
- **`disable`** (`Optional[List[str]]`) – List of pipeline components to disable See https://spacy.io/usage/processing-pipelines#disabling
- **`pre`** (`Optional[List[BaseMapper]]`) – Preprocessors to run before this preprocessor is executed
- **`memoize`** (`bool`) – Memoize preprocessor outputs?
- **`memoize_key`** (`Optional[Callable[[Any], Hashable]]`) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)
- **`gpu`** (`bool`) – Prefer Spacy GPU processing?

`__init__`(*text_field*, *doc_field*, *language='en_core_web_sm'*, *disable=None*, *pre=None*, *memoize=False*, *memoize_key=None*, *gpu=False*)

Initialize self. See help(type(self)) for accurate signature.

Return type `None`

Methods

`__init__`(text_field, doc_field[, language, ...])	Initialize self.
`reset_cache`()	Reset the memoization cache.

Continued on next page

Table 4 – continued from previous page

run(text)	Run the SpaCy model on input text.

__call__(*x*)
Run mapping function on input data point.

Deep copies the data point first so as not to make accidental in-place changes. If `memoize` is set to `True`, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.

Parameters **x** (`Any`) – Data point to run mapping function on

Returns Mapped data point of same format but possibly different fields

Return type DataPoint

reset_cache()
Reset the memoization cache.

Return type `None`

run(*text*)
Run the SpaCy model on input text.

Parameters **text** (`str`) – Text of document to parse

Returns Dictionary with a single key (`"doc"`), mapping to the parsed SpaCy `Doc` object

Return type FieldMap

6.5 snorkel.preprocess.spark.make_spark_preprocessor

snorkel.preprocess.spark.**make_spark_preprocessor**(*mapper*)
Convert `Mapper` to be compatible with PySpark.

Parameters **mapper** (`Mapper`) – Mapper to make compatible with PySpark

Return type `Mapper`

6.6 snorkel.preprocess.preprocessor

class snorkel.preprocess.**preprocessor**(*name=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Bases: `snorkel.map.core.lambda_mapper`

Decorate functions to create preprocessors.

See `snorkel.map.core.lambda_mapper` for details.

Example

```
>>> @preprocessor()
... def combine_text_preprocessor(x):
...     x.article = f"{x.title} {x.body}"
...     return x
>>> from snorkel.preprocess.nlp import SpacyPreprocessor
>>> spacy_preprocessor = SpacyPreprocessor("article", "article_parsed")
```

We can now add our preprocessors to an LF.

```
>>> preprocessors = [combine_text_preprocessor, spacy_preprocessor]
>>> from snorkel.labeling.lf import labeling_function
>>> @labeling_function(pre=preprocessors)
... def article_mentions_person(x):
...     for ent in x.article_parsed.ents:
...         if ent.label_ == "PERSON":
...             return ABSTAIN
...     return NEGATIVE
```

__init__(*name=None*, *pre=None*, *memoize=False*, *memoize_key=None*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, pre, memoize, memoize_key])	Initialize self.

__call__(*f*)
Wrap a function to create a LambdaMapper.

Parameters **f** (Callable[[Any], Optional[Any]]) – Function executing the mapping operation

Returns New LambdaMapper executing operation in wrapped function

Return type *LambdaMapper*

CHAPTER

SEVEN

SNORKEL SLICING PACKAGE

Programmatic data set slicing: SF creation, monitoring utilities, and representation learning for slices.

apply.dask.DaskSFApplier	SF applier for a Dask DataFrame.
sf.nlp.NLPSlicingFunction	Special labeling function type for spaCy-based LFs.
apply.dask.PandasParallelSFApplier	Parallel SF applier for a Pandas DataFrame.
PandasSFApplier	SF applier for a Pandas DataFrame.
SFApplier	SF applier for a list of data points.
SliceAwareClassifier	A slice-aware classifier that supports training + scoring on slice labels.
SliceCombinerModule	A module for combining the weighted representations learned by slices.
SlicingFunction	Base class for slicing functions.
apply.spark.SparkSFApplier	alias of *snorkel.labeling.apply.spark.SparkLFApplier*
add_slice_labels	Modify a dataloader in-place, adding labels for slice tasks.
convert_to_slice_tasks	Add slice labels to dataloader and creates new slice tasks (including base slice).
sf.nlp.nlp_slicing_function	Decorator to define a NLPSlicingFunction child object from a function.
slice_dataframe	Return a dataframe with examples corresponding to specified `SlicingFunction`.
slicing_function	Decorator to define a SlicingFunction object from a function.

7.1 snorkel.slicing.apply.dask.DaskSFApplier

class snorkel.slicing.apply.dask.**DaskSFApplier**(*lfs*)

Bases: *snorkel.labeling.apply.dask.DaskLFApplier*

SF applier for a Dask DataFrame.

See `snorkel.labeling.apply.dask.DaskLFApplier` for details.

__init__(*lfs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(df[, scheduler, fault_tolerant])	Label Dask DataFrame of data points with LFs.

apply (*df*, *scheduler='processes'*, *fault_tolerant=False*)
Label Dask DataFrame of data points with LFs.

Parameters

- **df** (*dask.dataframe.DataFrame*) – Dask DataFrame containing data points to be labeled by LFs
- **scheduler** (Union[str, dask.distributed.Client]) – A Dask scheduling configuration: either a string option or a Client. For more information, see https://docs.dask.org/en/stable/scheduling.html#
- **fault_tolerant** (bool) – Output -1 if LF execution fails?

Returns Matrix of labels emitted by LFs

Return type np.ndarray

7.2 snorkel.slicing.sf.nlp.NLPSlicingFunction

class snorkel.slicing.sf.nlp.**NLPSlicingFunction**(*name*, *f*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Bases: snorkel.labeling.lf.nlp.BaseNLPLabelingFunction

Special labeling function type for spaCy-based LFs.

This class is a special version of LabelingFunction. It has a SpacyPreprocessor integrated which shares a cache with all other NLPLabelingFunction instances. This makes it easy to define LFs that have a text input field and have logic written over spaCy Doc objects. Examples passed into an NLPLabelingFunction will have a new field which can be accessed which contains a spaCy Doc. By default, this field is called doc. A Doc object is a sequence of Token objects, which contain information on lemmatization, parts-of-speech, etc. Doc objects also contain fields like Doc.ents, a list of named entities, and Doc.noun_chunks, a list of noun phrases. For details of spaCy Doc objects and a full attribute listing, see https://spacy.io/api/doc.

Simple NLPLabelingFunctions can be defined via a decorator. See nlp_labeling_function.

Parameters

- **name** (str) – Name of the LF
- **f** (Callable[..., int]) – Function that implements the core LF logic
- **resources** (Optional[Mapping[str, Any]]) – Labeling resources passed in to f via kwargs
- **pre** (Optional[List[BaseMapper]]) – Preprocessors to run before SpacyPreprocessor is executed
- **text_field** (str) – Name of data point text field to input

- **doc_field** (str) – Name of data point field to output parsed document to
- **language** (str) – spaCy model to load See https://spacy.io/usage/models#usage
- **disable** (Optional[List[str]]) – List of pipeline components to disable See https://spacy.io/usage/processing-pipelines#disabling
- **memoize** (bool) – Memoize preprocessor outputs?
- **memoize_key** (Optional[Callable[[Any], Hashable]]) – Hashing function to handle the memoization (default to snorkel.map.core.get_hashable)

Raises **ValueError** – Calling incorrectly defined preprocessors

Example

```
>>> def f(x):
...     person_ents = [ent for ent in x.doc.ents if ent.label_ == "PERSON"]
...     return len(person_ents) > 0
>>> has_person_mention = NLPSlicingFunction(name="has_person_mention", f=f)
>>> has_person_mention
NLPSlicingFunction has_person_mention, Preprocessors: [SpacyPreprocessor...]
```

```
>>> from types import SimpleNamespace
>>> x = SimpleNamespace(text="The movie was good.")
>>> has_person_mention(x)
False
```

name
See above

__init__(*name*, *f*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, f[, resources, pre, ...])	Initialize self.

__call__(*x*)
Label data point.

Runs all preprocessors, then passes preprocessed data point to LF.

Parameters **x** (Any) – Data point to label

Returns Label for data point

Return type int

7.3 snorkel.slicing.apply.dask.PandasParallelSFApplier

class snorkel.slicing.apply.dask.**PandasParallelSFApplier**(*lfs*)

Bases: *snorkel.labeling.apply.dask.PandasParallelLFApplier*

Parallel SF applier for a Pandas DataFrame.

See `snorkel.labeling.apply.dask.PandasParallelLFApplier` for details.

__init__(*lfs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(df[, n_parallel, scheduler, ...])	Label Pandas DataFrame of data points with LFs in parallel using Dask.

apply(*df*, *n_parallel=2*, *scheduler='processes'*, *fault_tolerant=False*)

Label Pandas DataFrame of data points with LFs in parallel using Dask.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data points to be labeled by LFs
- **n_parallel** (int) – Parallelism level for LF application. Corresponds to `npartitions` in constructed Dask DataFrame. For `scheduler="processes"`, number of processes launched. Recommended to be no more than the number of cores on the running machine.
- **scheduler** (Union[str, dask.distributed.Client]) – A Dask scheduling configuration: either a string option or a `Client`. For more information, see https://docs.dask.org/en/stable/scheduling.html#
- **fault_tolerant** (bool) – Output `-1` if LF execution fails?

Returns Matrix of labels emitted by LFs

Return type np.ndarray

7.4 snorkel.slicing.PandasSFApplier

class snorkel.slicing.**PandasSFApplier**(*lfs*)

Bases: `snorkel.labeling.apply.pandas.PandasLFApplier`

SF applier for a Pandas DataFrame.

See `snorkel.labeling.core.PandasLFApplier` for details.

__init__(*lfs*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(df[, progress_bar, fault_tolerant, ...])	Label Pandas DataFrame of data points with LFs.

apply(*df*, *progress_bar=True*, *fault_tolerant=False*, *return_meta=False*)
Label Pandas DataFrame of data points with LFs.

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data points to be labeled by LFs
- **progress_bar** (bool) – Display a progress bar?
- **fault_tolerant** (bool) – Output −1 if LF execution fails?
- **return_meta** (bool) – Return metadata from apply call?

Return type Union[ndarray, Tuple[ndarray, ApplierMetadata]]

Returns

- *np.ndarray* – Matrix of labels emitted by LFs
- *ApplierMetadata* – Metadata, such as fault counts, for the apply call

7.5 snorkel.slicing.SFApplier

class snorkel.slicing.**SFApplier**(*lfs*)
Bases: snorkel.labeling.apply.core.LFApplier

SF applier for a list of data points.

See snorkel.labeling.core.LFApplier for details.

__init__(*lfs*)
Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(lfs)	Initialize self.
apply(data_points[, progress_bar, ...])	Label list of data points or a NumPy array with LFs.

apply(*data_points*, *progress_bar=True*, *fault_tolerant=False*, *return_meta=False*)
Label list of data points or a NumPy array with LFs.

Parameters

- **data_points** (Union[Sequence[Any], ndarray]) – List of data points or NumPy array to be labeled by LFs
- **progress_bar** (bool) – Display a progress bar?
- **fault_tolerant** (bool) – Output −1 if LF execution fails?
- **return_meta** (bool) – Return metadata from apply call?

Return type Union[ndarray, Tuple[ndarray, ApplierMetadata]]

Returns

- *np.ndarray* – Matrix of labels emitted by LFs
- *ApplierMetadata* – Metadata, such as fault counts, for the apply call

7.6 snorkel.slicing.SliceAwareClassifier

class snorkel.slicing.**SliceAwareClassifier**(*base_architecture*, *head_dim*, *slice_names*, *input_data_key='input_data'*, *task_name='task'*, *scorer=<snorkel.analysis.scorer.Scorer object>*, ***multitask_kwargs*)

Bases: snorkel.classification.multitask_classifier.MultitaskClassifier

A slice-aware classifier that supports training + scoring on slice labels.

NOTE: This model currently only supports binary classification.

Parameters

- **base_architecture** (Module) – A network architecture that accepts input data and outputs a representation
- **head_dim** (int) – Output feature dimension of the base_architecture, and input dimension of the internal prediction head: nn.Linear(head_dim, 2).
- **slice_names** (List[str]) – A list of slice names that the model will accept initialize as tasks and accept as corresponding labels
- **scorer** (Scorer) – A Scorer to be used for initialization of the MultitaskClassifier superclass.
- ****multitask_kwargs** – Arbitrary keyword arguments to be passed to the MultitaskClassifier superclass.

base_task
A base snorkel.classification.Task that the model will learn. This becomes a master_head_module that combines slice tasks information. For more, see snorkel.slicing.convert_to_slice_tasks.

slice_names
See above

__init__(*base_architecture*, *head_dim*, *slice_names*, *input_data_key='input_data'*, *task_name='task'*, *scorer=<snorkel.analysis.scorer.Scorer object>*, ***multitask_kwargs*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

Return type None

Methods

__init__(base_architecture, head_dim, . . . [, . . .])	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
add_task(task)	Add a single task to the network.

Continued on next page

Table 7 – continued from previous page

apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
calculate_loss(X_dict, Y_dict)	Calculate the loss for each task and the number of data points contributing.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(X_dict, task_names)	Do a forward pass through the network for all specified tasks.
half()	Casts all floating point parameters and buffers to half datatype.
load(model_path)	Load a saved model from the provided file path and moves it to a device.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
make_slice_dataloader(dataset, S, ...)	Create DictDataLoader with slice labels, initialized from specified dataset.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
predict(dataloader[, return_preds, remap_labels])	Calculate probabilities, (optionally) predictions, and pull out gold labels.
register_backward_hook(hook)	Registers a backward hook on the module.
register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
save(model_path)	Save the model to the specified file path.

Continued on next page

Table 7 – continued from previous page

score(dataloaders[, remap_labels, as_dataframe])	Calculate scores for the provided DictDataLoaders.
score_slices(dataloaders[, as_dataframe])	Scores appropriate slice labels using the overall prediction head.
share_memory()	**rtype** ~T
state_dict([destination, prefix, keep_vars])	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination
dump_patches

add_task(*task*)
Add a single task to the network.

Parameters **task** (Task) – A Task to add

Return type None

calculate_loss(*X_dict*, *Y_dict*)
Calculate the loss for each task and the number of data points contributing.

Parameters

- **X_dict** (Dict[str, Any]) – A dict of data fields
- **Y_dict** (Dict[str, Tensor]) – A dict from task names to label sets

Returns A dict of losses by task name and seen examples by task name

Return type Dict[str, torch.Tensor], Dict[str, float]

forward(*X_dict*, *task_names*)
Do a forward pass through the network for all specified tasks.

Parameters

- **X_dict** (Dict[str, Any]) – A dict of data fields
- **task_names** (Iterable[str]) – The names of the tasks to execute the forward pass for

Returns A dict mapping each operation name to its corresponding output

Return type OutputDict

Raises

- **TypeError** – If an Operation input has an invalid type
- **ValueError** – If a specified Operation failed to execute

load(*model_path*)
Load a saved model from the provided file path and moves it to a device.

Parameters **model_path** (str) – The path to a saved model

Return type None

make_slice_dataloader(*dataset*, *S*, ***dataloader_kwargs*)
Create DictDataLoader with slice labels, initialized from specified dataset.

Parameters

- **dataset** (DictDataset) – A DictDataset that will be converted into a slice-aware dataloader
- **S** (recarray) – A [num_examples, num_slices] slice matrix indicating whether each example is in every slice
- **slice_names** – A list of slice names corresponding to columns of S
- **dataloader_kwargs** (Any) – Arbitrary kwargs to be passed to DictDataLoader See DictDataLoader.__init__.

Return type DictDataLoader

predict(*dataloader*, *return_preds=False*, *remap_labels={}*)
Calculate probabilities, (optionally) predictions, and pull out gold labels.

Parameters

- **dataloader** (DictDataLoader) – A DictDataLoader to make predictions for
- **return_preds** (bool) – If True, include predictions in the return dict (not just probabilities)
- **remap_labels** (Dict[str, Optional[str]]) – A dict specifying which labels in the dataset's Y_dict (key) to remap to a new task (value)

Returns A dictionary mapping label type ('golds', 'probs', 'preds') to values

Return type Dict[str, Dict[str, torch.Tensor]]

save(*model_path*)
Save the model to the specified file path.

Parameters **model_path** (str) – The path where the model should be saved

Raises **BaseException** – If the torch.save() method fails

Return type None

score(*dataloaders*, *remap_labels={}*, *as_dataframe=False*)
Calculate scores for the provided DictDataLoaders.

Parameters

- **dataloaders** (List[DictDataLoader]) – A list of DictDataLoaders to calculate scores for
- **remap_labels** (Dict[str, Optional[str]]) – A dict specifying which labels in the dataset's Y_dict (key) to remap to a new task (value)
- **as_dataframe** (bool) – A boolean indicating whether to return results as pandas DataFrame (True) or dict (False)

Returns A dictionary mapping metric names to corresponding scores Metric names will be of the form "task/dataset/split/metric"

Return type Dict[str, float]

score_slices(*dataloaders*, *as_dataframe=False*)

Scores appropriate slice labels using the overall prediction head.

In other words, uses `base_task` (NOT `slice_tasks`) to evaluate slices.

In practice, we'd like to use a final prediction from a _single_ task head. To do so, `self.base_task` leverages reweighted slice representation to make a prediction. In this method, we remap all slice-specific `pred` labels to `self.base_task` for evaluation.

Parameters

- **dataloaders** (`List[DictDataLoader]`) – A list of DictDataLoaders to calculate scores for
- **as_dataframe** (`bool`) – A boolean indicating whether to return results as pandas DataFrame (True) or dict (False)
- **eval_slices_on_base_task** – A boolean indicating whether to remap slice labels to base task. Otherwise, keeps evaluation of slice labels on slice-specific heads.

Returns A dictionary mapping metric¡ names to corresponding scores Metric names will be of the form "task/dataset/split/metric"

Return type Dict[str, float]

7.7 snorkel.slicing.SliceCombinerModule

class snorkel.slicing.**SliceCombinerModule**(*slice_ind_key='_ind_head'*, *slice_pred_key='_pred_head'*, *slice_pred_feat_key='_pred_transform'*, *temperature=1.0*)

Bases: `torch.nn.modules.module.Module`

A module for combining the weighted representations learned by slices.

Intended for use with the MultitaskClassifier including:

- Indicator operations
- Prediction operations
- Prediction transform features

NOTE: This module currently only handles binary labels.

Parameters

- **slice_ind_key** (`str`) – Suffix of operation corresponding to the slice indicator heads
- **slice_pred_key** (`str`) – Suffix of operation corresponding to the slice predictor heads
- **slice_pred_feat_key** (`str`) – Suffix of operation corresponding to the slice predictor features heads
- **temperature** (`float`) – Temperature constant for scaling the weighting between indicator prediction and predictor confidences: SoftMax(indicator_pred * predictor_confidence / tau)

slice_ind_key

See above

slice_pred_key
See above

slice_pred_feat_key
See above

__init__(*slice_ind_key='_ind_head'*, *slice_pred_key='_pred_head'*, *slice_pred_feat_key='_pred_transform'*, *temperature=1.0*)
Initializes internal Module state, shared by both nn.Module and ScriptModule.

Return type None

Methods

__init__([slice_ind_key, slice_pred_key, ...])	Initializes internal Module state, shared by both nn.Module and ScriptModule.
add_module(name, module)	Adds a child module to the current module.
apply(fn)	Applies fn recursively to every submodule (as returned by .children()) as well as self.
bfloat16()	Casts all floating point parameters and buffers to bfloat16 datatype.
buffers([recurse])	Returns an iterator over module buffers.
children()	Returns an iterator over immediate children modules.
cpu()	Moves all model parameters and buffers to the CPU.
cuda([device])	Moves all model parameters and buffers to the GPU.
double()	Casts all floating point parameters and buffers to double datatype.
eval()	Sets the module in evaluation mode.
extra_repr()	Set the extra representation of the module
float()	Casts all floating point parameters and buffers to float datatype.
forward(output_dict)	Reweight and combine predictor representations given output dict.
half()	Casts all floating point parameters and buffers to half datatype.
load_state_dict(state_dict[, strict])	Copies parameters and buffers from state_dict into this module and its descendants.
modules()	Returns an iterator over all modules in the network.
named_buffers([prefix, recurse])	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
named_children()	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
named_modules([memo, prefix])	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
named_parameters([prefix, recurse])	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
parameters([recurse])	Returns an iterator over module parameters.
register_backward_hook(hook)	Registers a backward hook on the module.

Continued on next page

Table 9 – continued from previous page

register_buffer(name, tensor[, persistent])	Adds a buffer to the module.
register_forward_hook(hook)	Registers a forward hook on the module.
register_forward_pre_hook(hook)	Registers a forward pre-hook on the module.
register_full_backward_hook(hook)	Registers a backward hook on the module.
register_parameter(name, param)	Adds a parameter to the module.
requires_grad_([requires_grad])	Change if autograd should record operations on parameters in this module.
share_memory()	**rtype** ~T
state_dict([destination, prefix, keep_vars])	Returns a dictionary containing a whole state of the module.
to(*args, **kwargs)	Moves and/or casts the parameters and buffers.
train([mode])	Sets the module in training mode.
type(dst_type)	Casts all parameters and buffers to dst_type.
xpu([device])	Moves all model parameters and buffers to the XPU.
zero_grad([set_to_none])	Sets gradients of all model parameters to zero.

Attributes

T_destination
dump_patches

7.8 snorkel.slicing.SlicingFunction

class snorkel.slicing.**SlicingFunction**(*name*, *f*, *resources=None*, *pre=None*)

Bases: snorkel.labeling.lf.core.LabelingFunction

Base class for slicing functions.

See snorkel.labeling.lf.LabelingFunction for details.

__init__(*name*, *f*, *resources=None*, *pre=None*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__(name, f[, resources, pre])	Initialize self.

__call__(*x*)

Label data point.

Runs all preprocessors, then passes preprocessed data point to LF.

Parameters **x** (Any) – Data point to label

Returns Label for data point

Return type int

7.9 snorkel.slicing.apply.spark.SparkSFApplier

snorkel.slicing.apply.spark.**SparkSFApplier**
alias of *snorkel.labeling.apply.spark.SparkLFApplier*

7.10 snorkel.slicing.add_slice_labels

snorkel.slicing.**add_slice_labels**(*dataloader*, *base_task*, *S*)
Modify a dataloader in-place, adding labels for slice tasks.

Parameters

- **dataloader** (DictDataLoader) – A DictDataLoader whose dataset.Y_dict attribute will be modified in place
- **base_task** (Task) – The Task for which we want corresponding slice tasks/labels
- **S** (recarray) – A recarray (output of SFApplier) containing data fields with slice indicator information

Return type None

7.11 snorkel.slicing.convert_to_slice_tasks

snorkel.slicing.**convert_to_slice_tasks**(*base_task*, *slice_names*)
Add slice labels to dataloader and creates new slice tasks (including base slice).

Each slice will get two slice-specific heads: - an indicator head that learns to identify when DataPoints are in that slice - a predictor head that is trained on only members of that slice

The base task's head is replaced by a master head that makes predictions based on a combination of the predictor heads' predictions that are weighted by the indicator heads' prediction confidences.

WARNING: The current implementation pollutes the module_pool—the indicator task's module_pool includes predictor modules and vice versa since both are modified in place. This does not affect the result because the op sequences dictate which modules get used, and those do not include the extra modules. An alternative would be to make separate copies of the module pool for each, but that wastes time and memory extra copies of (potentially very large) modules that will be merged in a moment away in the model since they have the same name. We leave resolution of this issue for a future release.

Parameters

- **base_task** (Task) – Task for which we are adding slice tasks. As noted in the WARNING, this task's module_pool will currently be modified in place for efficiency purposes.
- **slice_names** (List[str]) – List of slice names corresponding to the columns of the slice matrix.

Returns Containins original base_task, pred/ind tasks for the base slice, and pred/ind tasks for each of the specified slice_names

Return type List[*Task*]

7.12 snorkel.slicing.sf.nlp.nlp_slicing_function

class snorkel.slicing.sf.nlp.**nlp_slicing_function**(*name=None*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Bases: snorkel.labeling.lf.nlp.base_nlp_labeling_function

Decorator to define a NLPSlicingFunction child object from a function.

TODO: Implement a common parent decorator for Snorkel operators

__init__(*name=None*, *resources=None*, *pre=None*, *text_field='text'*, *doc_field='doc'*, *language='en_core_web_sm'*, *disable=None*, *memoize=True*, *memoize_key=None*, *gpu=False*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, resources, pre, text_field, ...])	Initialize self.

__call__(*f*)

Wrap a function to create an BaseNLPLabelingFunction.

Parameters **f** (Callable[..., int]) – Function that implements the core NLP LF logic

Returns New BaseNLPLabelingFunction executing logic in wrapped function

Return type BaseNLPLabelingFunction

7.13 snorkel.slicing.slice_dataframe

snorkel.slicing.**slice_dataframe**(*df*, *slicing_function*)

Return a dataframe with examples corresponding to specified SlicingFunction.

Parameters

- **df** (DataFrame) – A pandas DataFrame that will be sliced
- **slicing_function** (SlicingFunction) – SlicingFunction which will operate over df to return a subset of examples; function returns a subset of data for which slicing_function output is True

Returns A DataFrame including only examples belonging to slice_name

Return type pd.DataFrame

7.14 snorkel.slicing.slicing_function

class snorkel.slicing.**slicing_function**(*name=None*, *resources=None*, *pre=None*)

Bases: object

Decorator to define a SlicingFunction object from a function.

Parameters

- **name** (Optional[str]) – Name of the SF
- **resources** (Optional[Mapping[str, Any]]) – Slicing resources passed in to f via kwargs
- **preprocessors** – Preprocessors to run on data points before SF execution

Examples

```
>>> @slicing_function()
... def f(x):
...     return x.a > 42
>>> f
SlicingFunction f, Preprocessors: []
>>> from types import SimpleNamespace
>>> x = SimpleNamespace(a=90, b=12)
>>> f(x)
True
```

```
>>> @slicing_function(name="my_sf")
... def g(x):
...     return 0 if x.a > 42 else -1
>>> g
SlicingFunction my_sf, Preprocessors: []
```

__init__(*name=None*, *resources=None*, *pre=None*)

Initialize self. See help(type(self)) for accurate signature.

Return type None

Methods

__init__([name, resources, pre])	Initialize self.

__call__(*f*)

Wrap a function to create a SlicingFunction.

Parameters **f** (Callable[..., int]) – Function that implements the core LF logic

Returns New SlicingFunction executing logic in wrapped function

Return type *SlicingFunction*

CHAPTER

EIGHT

SNORKEL UTILS PACKAGE

General machine learning utilities shared across Snorkel.

filter_labels	Filter out examples from arrays based on specified labels to filter.
preds_to_probs	Convert an array of predictions into an array of probabilistic labels.
probs_to_preds	Convert an array of probabilistic labels into an array of predictions.
to_int_label_array	Convert an array to a (possibly flattened) array of ints.

8.1 snorkel.utils.filter_labels

snorkel.utils.**filter_labels**(*label_dict*, *filter_dict*)

Filter out examples from arrays based on specified labels to filter.

The most common use of this method is to remove examples whose gold label is unknown (marked with a -1) or examples whose predictions were abstains (also -1) before calculating metrics.

NB: If an example matches the filter criteria for any label set, it will be removed from all label sets (so that the returned arrays are of the same size and still aligned).

Parameters

- **label_dict** (Dict[str, ndarray]) – A mapping from label set name to the array of labels The arrays in a label_dict.values() are assumed to be aligned
- **filter_dict** (Dict[str, List[int]]) – A mapping from label set name to the labels that should be filtered out for that label set

Returns A mapping with the same keys as label_dict but with filtered arrays as values

Return type Dict[str, np.ndarray]

Example

```
>>> golds = np.array([-1, 0, 0, 1, 0])
>>> preds = np.array([0, 0, 0, 1, -1])
>>> filtered = filter_labels(
...     label_dict={"golds": golds, "preds": preds},
...     filter_dict={"golds": [-1], "preds": [-1]}
... )
```

(continues on next page)

(continued from previous page)

```
>>> filtered["golds"]
array([0, 0, 1])
>>> filtered["preds"]
array([0, 0, 1])
```

8.2 snorkel.utils.preds_to_probs

snorkel.utils.**preds_to_probs**(*preds*, *num_classes*)

Convert an array of predictions into an array of probabilistic labels.

Parameters **pred** – A [num_datapoints] or [num_datapoints, 1] array of predictions

Returns A [num_datapoints, num_classes] array of probabilistic labels with probability of 1.0 in the column corresponding to the prediction

Return type np.ndarray

8.3 snorkel.utils.probs_to_preds

snorkel.utils.**probs_to_preds**(*probs*, *tie_break_policy='random'*, *tol=1e-05*)

Convert an array of probabilistic labels into an array of predictions.

Policies to break ties include: "abstain": return an abstain vote (-1) "true-random": randomly choose among the tied options "random": randomly choose among tied option using deterministic hash

NOTE: if tie_break_policy="true-random", repeated runs may have slightly different results due to difference in broken ties

Parameters

- **prob** – A [num_datapoints, num_classes] array of probabilistic labels such that each row sums to 1.
- **tie_break_policy** (str) – Policy to break ties when converting probabilistic labels to predictions
- **tol** (float) – The minimum difference among probabilities to be considered a tie

Returns A [n] array of predictions (integers in [0, . . . , num_classes - 1])

Return type np.ndarray

Examples

```
>>> probs_to_preds(np.array([[0.5, 0.5, 0.5]]), tie_break_policy="abstain")
array([-1])
>>> probs_to_preds(np.array([[0.8, 0.1, 0.1]]))
array([0])
```

8.4 snorkel.utils.to_int_label_array

snorkel.utils.**to_int_label_array**(*X*, *flatten_vector=True*)

Convert an array to a (possibly flattened) array of ints.

Cast all values to ints and possibly flatten [n, 1] arrays to [n]. This method is typically used to sanitize labels before use with analysis tools or metrics that expect 1D arrays as inputs.

Parameters

- **X** (ndarray) – An array to possibly flatten and possibly cast to int
- **flatten_vector** (bool) – If True, flatten array into a 1D array

Returns The converted array

Return type np.ndarray

Raises **ValueError** – Provided input could not be converted to an np.ndarray

INDEX

Symbols

A

B

I

K

L

M

N

O

P

R

S

T

U

W

X

Y