

---

# **Snorkel**

***Release 0.9.7***

**Snorkel Team**

**Mar 09, 2021**



## PACKAGE REFERENCE

<b>1 Snorkel Analysis Package</b>	<b>3</b>
<b>2 Snorkel Augmentation Package</b>	<b>7</b>
<b>3 Snorkel Classification Package</b>	<b>17</b>
<b>4 Snorkel Labeling Package</b>	<b>35</b>
<b>5 Snorkel Map Package</b>	<b>65</b>
<b>6 Snorkel Preprocess Package</b>	<b>71</b>
<b>7 Snorkel Slicing Package</b>	<b>77</b>
<b>8 Snorkel Utils Package</b>	<b>93</b>
<b>Index</b>	<b>97</b>



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](#).



## SNORKEL ANALYSIS PACKAGE

Generic model analysis utilities shared across Snorkel.

<i>Scorer</i>	Calculate one or more scores from user-specified and/or user-defined metrics.
<i>get_label_buckets</i>	Return data point indices bucketed by label combinations.
<i>get_label_instances</i>	Return instances in x with the specified combination of labels.
<i>metric_score</i>	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

<code>__init__</code> ([metrics, custom_metric_funcs, ...])	Initialize self.
<code>score</code> (golds[, preds, probs])	Calculate scores for one or more user-specified metrics.
<code>score_slices</code> (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** (reccarray) – A reccarray 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

## SNORKEL AUGMENTATION PACKAGE

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

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

---

<code>__init__(n_tfs[, n_per_original, keep_original])</code>	Initialize self.
<code>generate()</code>	Generate indices of all TFs in order.
<code>generate_for_example()</code>	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

<code>__init__(n_tfs[, keep_original])</code>	Initialize self.
<code>generate()</code>	Generate a sequence of TF indices.
<code>generate_for_example()</code>	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

<code>__init__([n_per_original, keep_original])</code>	Initialize self.
<code>generate()</code>	Generate indices of all TFs in order.
<code>generate_for_example()</code>	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

<code>__init__(n_tfs[, sequence_length, p, ...])</code>	Initialize self.
<code>generate()</code>	Generate a sequence of TF indices by sampling from distribution.
<code>generate_for_example()</code>	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

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

<code>__init__(n_tfs[, sequence_length, ...])</code>	Initialize self.
<code>generate()</code>	Generate a sequence of TF indices by sampling from distribution.
<code>generate_for_example()</code>	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

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

<code>__init__(name[, field_names, ...])</code>	Initialize self.
<code>reset_cache()</code>	Reset the memoization cache.
<code>run(**kwargs)</code>	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`



## SNORKEL CLASSIFICATION PACKAGE

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

<i>Checkpointner</i>	Manager for checkpointing model.
<i>CheckpointnerConfig</i>	Manager for checkpointing model.
<i>DictDataLoader</i>	A DataLoader that uses the appropriate collate_fn for a DictDataset.
<i>DictDataset</i>	A dataset where both the data fields and labels are stored in as dictionaries.
<i>LogManager</i>	A class to manage logging during training progress.
<i>LogManagerConfig</i>	Manager for checkpointing model.
<i>LogWriter</i>	A class for writing logs.
<i>LogWriterConfig</i>	Manager for checkpointing model.
<i>MultitaskClassifier</i>	A classifier built from one or more tasks to support advanced workflows.
<i>Operation</i>	A single operation (forward pass of a module) to execute in a Task.
<i>Task</i>	A single task (a collection of modules and specified path through them).
<i>TensorBoardWriter</i>	A class for logging to Tensorboard during training process.
<i>Trainer</i>	A class for training a MultitaskClassifier.
<i>cross_entropy_with_probs</i>	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 LogManagerer
- **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

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

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

## Attributes

<code>checkpoint_clear</code>	Alias for field number 5
<code>checkpoint_dir</code>	Alias for field number 0
<code>checkpoint_factor</code>	Alias for field number 1
<code>checkpoint_metric</code>	Alias for field number 2
<code>checkpoint_runway</code>	Alias for field number 4
<code>checkpoint_task_metrics</code>	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 overridden 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

---

<code>__init__</code> ( <i>dataset</i> [, <i>collate_fn</i> ])	Initialize self.
<code>check_worker_number_rationality</code> ()	

---

#### Attributes

---

<code>multiprocessing_context</code>
--------------------------------------

---

## 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

<code>__init__</code> ( <i>name, split, X_dict, Y_dict</i> )	Initialize self.
<code>from_tensors</code> ( <i>X_tensor, Y_tensor, split[, ...]</i> )	Initialize a <code>DictDataset</code> 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

<code>__init__(n_batches_per_epoch[, log_writer, ...])</code>	Initialize self.
<code>cleanup(model)</code>	Close the log writer and checkpointer if needed.
<code>reset()</code>	Reset counters.
<code>trigger_checkpointing()</code>	Check if current counts trigger checkpointing.
<code>trigger_evaluation()</code>	Check if current counts trigger evaluation.
<code>update(batch_size)</code>	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**

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

## Attributes

<code>counter_unit</code>	Alias for field number 0
<code>evaluation_freq</code>	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

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

**Parameters**

- **dict\_to\_write** (*Mapping[str, Any]*) – JSON-compatible 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

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

## Attributes

<code>log_dir</code>	Alias for field number 0
<code>run_name</code>	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**

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

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

---

<code>__init__</code> ( <i>module_name</i> , <i>inputs</i> [, <i>name</i> ])	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 probabilistic 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

<code>__init__(**kwargs)</code>	Initialize self.
<code>add_scalar(name, value, step)</code>	Log a scalar variable to TensorBoard.
<code>cleanup()</code>	Close the SummaryWriter.
<code>write_config(config[, config_filename])</code>	Dump the config to file and add it to TensorBoard.
<code>write_json(dict_to_write, filename)</code>	Dump a JSON-compatible object to root log directory.
<code>write_log(log_filename)</code>	Dump the scalar value log to file.
<code>write_text(text, filename)</code>	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-compatible object to root log directory.

**Parameters**

- **dict\_to\_write** (`Mapping[str, Any]`) – JSON-compatible 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**

<code>__init__</code> ( <i>[name]</i> )	Initialize self.
<code>fit</code> ( <i>model, dataloaders</i> )	Train a MultitaskClassifier.
<code>load</code> ( <i>trainer_path, model</i> )	Load trainer config and optimizer state from the specified json file path to the trainer object.
<code>save</code> ( <i>trainer_path</i> )	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 not 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

## SNORKEL LABELING PACKAGE

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

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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(df[, scheduler, fault_tolerant])</code>	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 *j*th LF to the *i*th 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

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

Continued on next page



Table 3 – continued from previous page

<code>label_overlap()</code>	Compute the fraction of data points with at least two (non-abstain) labels.
<code>lf_conflicts([normalize_by_overlaps])</code>	Compute frac.
<code>lf_coverages()</code>	Compute frac.
<code>lf_empirical_accuracies(Y)</code>	Compute empirical accuracy against a set of labels Y for each LF.
<code>lf_empirical_probs(Y, k)</code>	Estimate conditional probability tables for each LF.
<code>lf_overlaps([normalize_by_coverage])</code>	Compute frac.
<code>lf_polarities()</code>	Infer the polarities of each LF based on evidence in a label matrix.
<code>lf_summary([Y, est_weights])</code>	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],
... ])
>>> LFAalysis(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],
... ])
>>> LFAalysis(L).lf_conflicts()
array([0.2, 0.2, 0. ])
>>> LFAalysis(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],
... ])
>>> LFAalysis(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  $\{1, \dots, k\}$
- *k* (int) – The cardinality i.e. number of classes

**Returns** An  $m \times (k+1) \times k$  np.ndarray representing the  $m$   $(k+1) \times k$  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],
... ])
>>> LFAalysis(L).lf_overlaps()
array([0.4, 0.6, 0.4])
>>> LFAalysis(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],
... ])
>>> LFAalysis(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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(data_points[, progress_bar, ...])</code>	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

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

Continued on next page

Table 5 – continued from previous page

<code>get_conditional_probs()</code>	Return the estimated conditional probabilities table.
<code>get_weights()</code>	Return the vector of learned LF weights for combining LFs.
<code>half()</code>	Casts all floating point parameters and buffers to <code>half</code> datatype.
<code>load(source)</code>	Load existing label model.
<code>load_state_dict(state_dict[, strict])</code>	Copies parameters and buffers from <code>state_dict</code> into this module and its descendants.
<code>modules()</code>	Returns an iterator over all modules in the network.
<code>named_buffers([prefix, recurse])</code>	Returns an iterator over module buffers, yielding both the name of the buffer as well as the buffer itself.
<code>named_children()</code>	Returns an iterator over immediate children modules, yielding both the name of the module as well as the module itself.
<code>named_modules([memo, prefix])</code>	Returns an iterator over all modules in the network, yielding both the name of the module as well as the module itself.
<code>named_parameters([prefix, recurse])</code>	Returns an iterator over module parameters, yielding both the name of the parameter as well as the parameter itself.
<code>parameters([recurse])</code>	Returns an iterator over module parameters.
<code>predict(L[, return_probs, tie_break_policy])</code>	Return predicted labels, with ties broken according to policy.
<code>predict_proba(L)</code>	Return label probabilities $P(Y   \lambda)$ .
<code>register_backward_hook(hook)</code>	Registers a backward hook on the module.
<code>register_buffer(name, tensor[, persistent])</code>	Adds a buffer to the module.
<code>register_forward_hook(hook)</code>	Registers a forward hook on the module.
<code>register_forward_pre_hook(hook)</code>	Registers a forward pre-hook on the module.
<code>register_full_backward_hook(hook)</code>	Registers a backward hook on the module.
<code>register_parameter(name, param)</code>	Adds a parameter to the module.
<code>requires_grad_([requires_grad])</code>	Change if autograd should record operations on parameters in this module.
<code>save(destination)</code>	Save label model.
<code>score(L, Y[, metrics, tie_break_policy])</code>	Calculate one or more scores from user-specified and/or user-defined metrics.
<code>share_memory()</code>	
	<b>rtype ~T</b>
<code>state_dict([destination, prefix, keep_vars])</code>	Returns a dictionary containing a whole state of the module.
<code>to(*args, **kwargs)</code>	Moves and/or casts the parameters and buffers.
<code>train([mode])</code>	Sets the module in training mode.
<code>type(dst_type)</code>	Casts all parameters and buffers to <code>dst_type</code> .
<code>xpu([device])</code>	Moves all model parameters and buffers to the XPU.
<code>zero_grad([set_to_none])</code>	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 probabilities table `cprobs`, where `cprobs` is an  $(m, k+1, k)$ -dim `np.ndarray` with:

$$cprobs[i, j, k] = P(lf_i = j-1 \mid Y = k)$$

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

**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}

**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  $\{-1,0,1,\dots,k-1\}$
- **Y** (`ndarray`) – Gold labels associated with data points in `L`
- **metrics** (`Optional[List[str]]`) – A list of metric names. Possible 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**

---

<code>__init__(name, f[, resources, pre])</code>	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**

---

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

---

Continued on next page

Table 8 – continued from previous page

<code>score(L, Y[, metrics, tie_break_policy])</code>	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

<code>__init__</code> ([cardinality])	Initialize self.
<code>load</code> (source)	Load existing label model.
<code>predict</code> (L[, return_probs, tie_break_policy])	Return predicted labels, with ties broken according to policy.
<code>predict_proba</code> (L)	Predict probabilities using majority vote.
<code>save</code> (destination)	Save label model.
<code>score</code> (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

---

<code>__init__(name, f[, resources, pre, ...])</code>	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

---

<code>__init__(lfs)</code>	Initialize self.
<code>apply(df[, progress_bar, fault_tolerant, ...])</code>	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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(df[, n_parallel, scheduler, ...])</code>	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

<code>__init__</code> ( <i>cardinality</i> )	Initialize self.
<code>load</code> ( <i>source</i> )	Load existing label model.
<code>predict</code> ( <i>L</i> , <i>return_probs</i> , <i>tie_break_policy</i> )	Return predicted labels, with ties broken according to policy.
<code>predict_proba</code> ( <i>L</i> )	Assign random votes to the data points.
<code>save</code> ( <i>destination</i> )	Save label model.
<code>score</code> ( <i>L</i> , <i>Y</i> [, <i>metrics</i> , <i>tie_break_policy</i> ])	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  $\{-1,0,1,\dots,k-1\}$
- **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  $\{-1,0,1,\dots,k-1\}$
- **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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(data_points[, fault_tolerant])</code>	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



---

<code>__init__</code> ([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,
                                                                re-
                                                                sources=None,
                                                                pre=None,
                                                                text_field='text',
                                                                doc_field='doc',
                                                                lan-
                                                                guage='en_core_web_sm',
                                                                disable=None,
                                                                memo-
                                                                ize=True,
                                                                memo-
                                                                ize_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`



## SNORKEL MAP PACKAGE

Generic utilities for data point to data point operations.

<i>BaseMapper</i>	Base class for Mapper and LambdaMapper.
<i>LambdaMapper</i>	Define a mapper from a function.
<i>Mapper</i>	Base class for any data point to data point mapping in the pipeline.
<i>lambda_mapper</i>	Decorate a function to define a LambdaMapper object.
<i>spark.make_spark_mapper</i>	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

---

<code>__init__(name, pre, memoize[, memoize_key])</code>	Initialize self.
<code>reset_cache()</code>	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

---

<code>__init__(name, f[, pre, memoize, memoize_key])</code>	Initialize self.
<code>reset_cache()</code>	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**

<code>__init__(name[, field_names, ...])</code>	Initialize self.
<code>reset_cache()</code>	Reset the memoization cache.
<code>run(**kwargs)</code>	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`



## SNORKEL PREPROCESS PACKAGE

Preprocessors for LFs, TFs, and SFs.

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

<code>__init__(name, f[, pre, memoize, memoize_key])</code>	Initialize self.
<code>reset_cache()</code>	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**  $\mathbf{x}$  (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

<code>__init__(name[, field_names, ...])</code>	Initialize self.
<code>reset_cache()</code>	Reset the memoization cache.
<code>run(**kwargs)</code>	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**  $\mathbf{x}$  (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,      lan-
                                             guage='en_core_web_sm',      dis-
                                             able=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, memo-
         ize=False, memoize_key=None, gpu=False)
```

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

**Return type** None

### Methods

<code>__init__</code> (text_field, doc_field[, language, ...])	Initialize self.
<code>reset_cache</code> ()	Reset the memoization cache.

Continued on next page

Table 4 – continued from previous page

<code>run(text)</code>	Run the SpaCy model on input text.
<code>__call__(x)</code>	<p>Run mapping function on input data point.</p> <p>Deep copies the data point first so as not to make accidental in-place changes. If <code>memoize</code> is set to <code>True</code>, an internal cache is checked for results. If no cached results are found, the computed results are added to the cache.</p> <p><b>Parameters</b> <code>x</code> (Any) – Data point to run mapping function on</p> <p><b>Returns</b> Mapped data point of same format but possibly different fields</p> <p><b>Return type</b> <code>DataPoint</code></p>
<code>reset_cache()</code>	<p>Reset the memoization cache.</p> <p><b>Return type</b> <code>None</code></p>
<code>run(text)</code>	<p>Run the SpaCy model on input text.</p> <p><b>Parameters</b> <code>text</code> (<code>str</code>) – Text of document to parse</p> <p><b>Returns</b> Dictionary with a single key ("<code>doc</code>"), mapping to the parsed SpaCy <code>Doc</code> object</p> <p><b>Return type</b> <code>FieldMap</code></p>

## 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*





## SNORKEL SLICING PACKAGE

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

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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(df[, scheduler, fault_tolerant])</code>	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 `NLPLabelingFunction`s 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

<code>__init__</code> ( <i>lfs</i> )	Initialize self.
<code>apply</code> ( <i>df</i> , <i>n_parallel</i> , <i>scheduler</i> , ...) ]	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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(df[, progress_bar, fault_tolerant, ...])</code>	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

<code>__init__(lfs)</code>	Initialize self.
<code>apply(data_points[, progress_bar, ...])</code>	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**

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

Continued on next page

Table 7 – continued from previous page

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

Continued on next page

Table 7 – continued from previous page

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

### Attributes

<code>T_destination</code>
<code>dump_patches</code>

#### **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\_data\_loader** (*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:  $\text{SoftMax}(\text{indicator\_pred} * \text{predictor\_confidence} / \text{tau})$

**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**

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

Continued on next page

Table 9 – continued from previous page

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

---

<code>__init__</code> ([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*





## SNORKEL UTILS PACKAGE

General machine learning utilities shared across Snorkel.

<code>filter_labels</code>	Filter out examples from arrays based on specified labels to filter.
<code>preds_to_probs</code>	Convert an array of predictions into an array of probabilistic labels.
<code>probs_to_preds</code>	Convert an array of probabilistic labels into an array of predictions.
<code>to_int_label_array</code>	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`



## Symbols

- `__call__()` (*snorkel.augmentation.TransformationFunction method*), 14
- `__call__()` (*snorkel.augmentation.transformation\_function method*), 15
- `__call__()` (*snorkel.labeling.LabelingFunction method*), 48
- `__call__()` (*snorkel.labeling.labeling\_function method*), 61
- `__call__()` (*snorkel.labeling.lf.nlp.NLPLabelingFunction method*), 54
- `__call__()` (*snorkel.labeling.lf.nlp.nlp\_labeling\_function method*), 62
- `__call__()` (*snorkel.labeling.lf.nlp\_spark.SparkNLPLabelingFunction method*), 59
- `__call__()` (*snorkel.labeling.lf.nlp\_spark.spark\_nlp\_labeling\_function method*), 63
- `__call__()` (*snorkel.map.BaseMapper method*), 66
- `__call__()` (*snorkel.map.LambdaMapper method*), 66
- `__call__()` (*snorkel.map.Mapper method*), 68
- `__call__()` (*snorkel.map.lambda\_mapper method*), 69
- `__call__()` (*snorkel.preprocess.LambdaPreprocessor method*), 71
- `__call__()` (*snorkel.preprocess.Preprocessor method*), 72
- `__call__()` (*snorkel.preprocess.nlp.SpacyPreprocessor method*), 74
- `__call__()` (*snorkel.preprocess.preprocessor method*), 75
- `__call__()` (*snorkel.slicing.SlicingFunction method*), 88
- `__call__()` (*snorkel.slicing.sf.nlp.NLPSlicingFunction method*), 79
- `__call__()` (*snorkel.slicing.sf.nlp.nlp\_slicing\_function method*), 90
- `__call__()` (*snorkel.slicing.slicing\_function method*), 91
- `__init__()` (*snorkel.analysis.Scorer method*), 3
- `__init__()` (*snorkel.augmentation.ApplyAllPolicy method*), 8
- `__init__()` (*snorkel.augmentation.ApplyEachPolicy method*), 9
- `__init__()` (*snorkel.augmentation.ApplyOnePolicy method*), 9
- `__init__()` (*snorkel.augmentation.MeanFieldPolicy method*), 10
- `__init__()` (*snorkel.augmentation.PandasTFApplier method*), 11
- `__init__()` (*snorkel.augmentation.RandomPolicy method*), 12
- `__init__()` (*snorkel.augmentation.TFApplier method*), 13
- `__init__()` (*snorkel.augmentation.TransformationFunction method*), 14
- `__init__()` (*snorkel.augmentation.transformation\_function method*), 15
- `__init__()` (*snorkel.classification.Checkpointer method*), 17
- `__init__()` (*snorkel.classification.CheckpointerConfig method*), 19
- `__init__()` (*snorkel.classification.DictDataLoader method*), 20
- `__init__()` (*snorkel.classification.DictDataset method*), 20
- `__init__()` (*snorkel.classification.LogManager method*), 21
- `__init__()` (*snorkel.classification.LogManagerConfig method*), 22
- `__init__()` (*snorkel.classification.LogWriter method*), 23
- `__init__()` (*snorkel.classification.LogWriterConfig method*), 25
- `__init__()` (*snorkel.classification.MultitaskClassifier method*), 26
- `__init__()` (*snorkel.classification.Operation method*), 29
- `__init__()` (*snorkel.classification.Task method*), 30
- `__init__()` (*snorkel.classification.TensorBoardWriter method*), 31
- `__init__()` (*snorkel.classification.Trainer method*), 33
- `__init__()` (*snorkel.labeling.LFAnalysis method*), 36

[\\_\\_init\\_\\_\(\) \(snorkel.labeling.LFApplier method\), 41](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.LabelingFunction method\), 48](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.PandasLFApplier method\), 54](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.apply.dask.DaskLFApplier method\), 35](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.apply.dask.PandasParallelLFApplier method\), 55](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.apply.spark.SparkLFApplier method\), 58](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.labeling\\_function method\), 60](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.lf.nlp.NLPLabelingFunction method\), 53](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.lf.nlp.nlp\\_labeling\\_function method\), 62](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.lf.nlp.spark.SparkNLPLabelingFunction method\), 59](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.lf.nlp.spark.spark\\_nlp\\_labeling\\_function method\), 63](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.model.baselines.MajorityClassVoter method\), 48](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.model.baselines.MajorityLabelVoter method\), 50](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.model.baselines.RandomVoter method\), 56](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.labeling.model.label\\_model.LabelModel method\), 42](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.map.BaseMapper method\), 65](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.map.LambdaMapper method\), 66](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.map.Mapper method\), 68](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.map.lambda\\_mapper method\), 69](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.preprocess.LambdaPreprocessor method\), 71](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.preprocess.Preprocessor method\), 72](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.preprocess.nlp.SpacyPreprocessor method\), 73](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.preprocess.preprocessor method\), 75](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.PandasSFApplier method\), 80](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.SFApplier method\), 81](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.SliceAwareClassifier method\), 82](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.SliceCombinerModule method\), 87](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.SlicingFunction method\), 88](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.apply.dask.DaskSFApplier method\), 77](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.apply.dask.PandasParallelSFApplier method\), 80](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.sf.nlp.NLPSlicingFunction method\), 79](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.sf.nlp.nlp\\_slicing\\_function method\), 90](#)  
[\\_\\_init\\_\\_\(\) \(snorkel.slicing.slicing\\_function method\), 91](#)

## A

[add\\_scalar\(\) \(snorkel.classification.LogWriter method\), 24](#)  
[add\\_scalar\(\) \(snorkel.classification.TensorBoardWriter method\), 31](#)  
[add\\_slice\\_labels\(\) \(in module snorkel.slicing\), 89](#)  
[add\\_task\(\) \(snorkel.classification.MultitaskClassifier method\), 27](#)  
[add\\_task\(\) \(snorkel.slicing.SliceAwareClassifier method\), 84](#)  
[apply\(\) \(snorkel.augmentation.PandasTFApplier method\), 11](#)  
[apply\(\) \(snorkel.augmentation.TFApplier method\), 13](#)  
[apply\(\) \(snorkel.labeling.apply.dask.DaskLFApplier method\), 36](#)  
[apply\(\) \(snorkel.labeling.apply.dask.PandasParallelLFApplier method\), 55](#)  
[apply\(\) \(snorkel.labeling.apply.spark.SparkLFApplier method\), 58](#)  
[apply\(\) \(snorkel.labeling.LFApplier method\), 41](#)  
[apply\(\) \(snorkel.labeling.PandasLFApplier method\), 54](#)  
[apply\(\) \(snorkel.slicing.apply.dask.DaskSFApplier method\), 78](#)  
[apply\(\) \(snorkel.slicing.apply.dask.PandasParallelSFApplier method\), 80](#)  
[apply\(\) \(snorkel.slicing.PandasSFApplier method\), 81](#)  
[apply\(\) \(snorkel.slicing.SFApplier method\), 81](#)  
[apply\\_generator\(\) \(snorkel.augmentation.PandasTFApplier method\), 12](#)  
[apply\\_generator\(\) \(snorkel.augmentation.TFApplier method\), 13](#)  
[ApplyAllPolicy \(class in snorkel.augmentation\), 7](#)  
[ApplyEachPolicy \(class in snorkel.augmentation\), 8](#)  
[ApplyOnePolicy \(class in snorkel.augmentation\), 9](#)

## B

[base\\_task \(snorkel.slicing.SliceAwareClassifier attribute\), 82](#)  
[BaseMapper \(class in snorkel.map\), 65](#)  
[BasePreprocessor \(in module snorkel.preprocess\), 71](#)

- batch\_scheduler (*snorkel.classification.Trainer* attribute), 32
- ## C
- calculate\_loss() (*snorkel.classification.MultitaskClassifier* method), 27
- calculate\_loss() (*snorkel.slicing.SliceAwareClassifier* method), 84
- cardinality (*snorkel.labeling.model.label\_model.LabelModel* attribute), 42
- checkpoint() (*snorkel.classification.Checkpointer* method), 18
- checkpoint\_clear() (*snorkel.classification.CheckpointerConfig* property), 19
- checkpoint\_dir() (*snorkel.classification.CheckpointerConfig* property), 19
- checkpoint\_factor() (*snorkel.classification.CheckpointerConfig* property), 19
- checkpoint\_metric() (*snorkel.classification.CheckpointerConfig* property), 19
- checkpoint\_runway() (*snorkel.classification.CheckpointerConfig* property), 19
- checkpoint\_task\_metrics() (*snorkel.classification.CheckpointerConfig* property), 19
- Checkpointier (class in *snorkel.classification*), 17
- checkpointer (*snorkel.classification.Trainer* attribute), 32
- CheckpointierConfig (class in *snorkel.classification*), 18
- cleanup() (*snorkel.classification.LogManager* method), 22
- cleanup() (*snorkel.classification.LogWriter* method), 24
- cleanup() (*snorkel.classification.TensorBoardWriter* method), 31
- clear() (*snorkel.classification.Checkpointer* method), 18
- config (*snorkel.classification.LogWriter* attribute), 23
- config (*snorkel.classification.MultitaskClassifier* attribute), 25
- config (*snorkel.classification.Trainer* attribute), 32
- config (*snorkel.labeling.model.label\_model.LabelModel* attribute), 42
- convert\_to\_slice\_tasks() (in module *snorkel.slicing*), 89
- count() (*snorkel.classification.CheckpointerConfig* method), 19
- count() (*snorkel.classification.LogManagerConfig* method), 23
- count() (*snorkel.classification.LogWriterConfig* method), 25
- counter\_unit() (*snorkel.classification.LogManagerConfig* property), 23
- cross\_entropy\_with\_probs() (in module *snorkel.classification*), 34
- ## D
- DaskLFApplier (class in *snorkel.labeling.apply.dask*), 35
- DaskSFApplier (class in *snorkel.slicing.apply.dask*), 77
- DictDataLoader (class in *snorkel.classification*), 19
- DictDataset (class in *snorkel.classification*), 20
- ## E
- evaluation\_freq() (*snorkel.classification.LogManagerConfig* property), 23
- ## F
- field\_names (*snorkel.map.Mapper* attribute), 67
- filter\_labels() (in module *snorkel.utils*), 93
- filter\_unlabeled\_dataframe() (in module *snorkel.labeling*), 59
- fit() (*snorkel.classification.Trainer* method), 33
- fit() (*snorkel.labeling.model.baselines.MajorityClassVoter* method), 49
- fit() (*snorkel.labeling.model.label\_model.LabelModel* method), 44
- forward() (*snorkel.classification.MultitaskClassifier* method), 27
- forward() (*snorkel.labeling.model.label\_model.LabelModel* method), 44
- forward() (*snorkel.slicing.SliceAwareClassifier* method), 84
- from\_tensors() (*snorkel.classification.DictDataset* class method), 21
- ## G
- generate() (*snorkel.augmentation.ApplyAllPolicy* method), 8
- generate() (*snorkel.augmentation.ApplyEachPolicy* method), 9
- generate() (*snorkel.augmentation.ApplyOnePolicy* method), 10
- generate() (*snorkel.augmentation.MeanFieldPolicy* method), 11
- generate() (*snorkel.augmentation.RandomPolicy* method), 13
- generate\_for\_example() (*snorkel.augmentation.ApplyAllPolicy* method), 8

`generate_for_example()` (*snorkel.augmentation.ApplyEachPolicy method*), 9  
`generate_for_example()` (*snorkel.augmentation.ApplyOnePolicy method*), 10  
`generate_for_example()` (*snorkel.augmentation.MeanFieldPolicy method*), 11  
`generate_for_example()` (*snorkel.augmentation.RandomPolicy method*), 13  
`get_conditional_probs()` (*snorkel.labeling.model.label\_model.LabelModel method*), 45  
`get_label_buckets()` (*in module snorkel.analysis*), 5  
`get_label_instances()` (*in module snorkel.analysis*), 5  
`get_weights()` (*snorkel.labeling.model.label\_model.LabelModel method*), 45

**I**

`index()` (*snorkel.classification.CheckpointerConfig method*), 19  
`index()` (*snorkel.classification.LogManagerConfig method*), 23  
`index()` (*snorkel.classification.LogWriterConfig method*), 25  
`inputs` (*snorkel.classification.Operation attribute*), 29

**K**

`keep_original` (*snorkel.augmentation.ApplyAllPolicy attribute*), 8  
`keep_original` (*snorkel.augmentation.ApplyEachPolicy attribute*), 9  
`keep_original` (*snorkel.augmentation.MeanFieldPolicy attribute*), 10  
`keep_original` (*snorkel.augmentation.RandomPolicy attribute*), 12

**L**

`L` (*snorkel.labeling.LFAnalysis attribute*), 36  
`label_conflict()` (*snorkel.labeling.LFAnalysis method*), 37  
`label_coverage()` (*snorkel.labeling.LFAnalysis method*), 37  
`label_overlap()` (*snorkel.labeling.LFAnalysis method*), 37  
`labeling_function` (*class in snorkel.labeling*), 60  
`LabelingFunction` (*class in snorkel.labeling*), 47  
`LabelModel` (*class in snorkel.labeling.model.label\_model*), 41  
`lambda_mapper` (*class in snorkel.map*), 68  
`LambdaMapper` (*class in snorkel.map*), 66  
`LambdaPreprocessor` (*class in snorkel.preprocess*), 71  
`lf_conflicts()` (*snorkel.labeling.LFAnalysis method*), 38  
`lf_coverages()` (*snorkel.labeling.LFAnalysis method*), 38  
`lf_empirical_accuracies()` (*snorkel.labeling.LFAnalysis method*), 39  
`lf_empirical_probs()` (*snorkel.labeling.LFAnalysis method*), 39  
`lf_overlaps()` (*snorkel.labeling.LFAnalysis method*), 39  
`lf_polarities()` (*snorkel.labeling.LFAnalysis method*), 39  
`lf_summary()` (*snorkel.labeling.LFAnalysis method*), 40  
`LFAnalysis` (*class in snorkel.labeling*), 36  
`LFApplier` (*class in snorkel.labeling*), 40  
`LabelModel` (*snorkel.classification.MultitaskClassifier method*), 28  
`load()` (*snorkel.classification.Trainer method*), 33  
`load()` (*snorkel.labeling.model.baselines.MajorityClassVoter method*), 49  
`load()` (*snorkel.labeling.model.baselines.MajorityLabelVoter method*), 51  
`load()` (*snorkel.labeling.model.baselines.RandomVoter method*), 56  
`load()` (*snorkel.labeling.model.label\_model.LabelModel method*), 45  
`load()` (*snorkel.slicing.SliceAwareClassifier method*), 84  
`load_best_model()` (*snorkel.classification.Checkpointer method*), 18  
`log_dir` (*snorkel.classification.LogWriter attribute*), 23  
`log_dir()` (*snorkel.classification.LogWriterConfig property*), 25  
`log_manager` (*snorkel.classification.Trainer attribute*), 32  
`log_writer` (*snorkel.classification.Trainer attribute*), 32  
`LogManager` (*class in snorkel.classification*), 21  
`LogManagerConfig` (*class in snorkel.classification*), 22  
`LogWriter` (*class in snorkel.classification*), 23  
`LogWriterConfig` (*class in snorkel.classification*), 24  
`loss_func` (*snorkel.classification.Task attribute*), 30  
`loss_funcs` (*snorkel.classification.MultitaskClassifier attribute*), 25  
`lr_scheduler` (*snorkel.classification.Trainer attribute*), 32



## M

MajorityClassVoter (class in *snorkel.labeling.model.baselines*), 48

MajorityLabelVoter (class in *snorkel.labeling.model.baselines*), 50

make\_slice\_data\_loader() (*snorkel.slicing.SliceAwareClassifier* method), 85

make\_spark\_mapper() (in module *snorkel.map.spark*), 69

make\_spark\_preprocessor() (in module *snorkel.preprocess.spark*), 74

mapped\_field\_names (*snorkel.map.Mapper* attribute), 67

Mapper (class in *snorkel.map*), 67

MeanFieldPolicy (class in *snorkel.augmentation*), 10

memoize (*snorkel.map.BaseMapper* attribute), 65

memoize (*snorkel.map.lambda\_mapper* attribute), 69

memoize (*snorkel.map.Mapper* attribute), 68

metric\_score() (in module *snorkel.analysis*), 6

metrics (*snorkel.analysis.Scorer* attribute), 3

module\_name (*snorkel.classification.Operation* attribute), 29

module\_pool (*snorkel.classification.MultitaskClassifier* attribute), 25

module\_pool (*snorkel.classification.Task* attribute), 30

MultitaskClassifier (class in *snorkel.classification*), 25

## N

n (*snorkel.augmentation.ApplyAllPolicy* attribute), 7

n (*snorkel.augmentation.ApplyEachPolicy* attribute), 8

n (*snorkel.augmentation.MeanFieldPolicy* attribute), 10

n (*snorkel.augmentation.RandomPolicy* attribute), 12

n\_per\_original (*snorkel.augmentation.ApplyAllPolicy* attribute), 7

n\_per\_original (*snorkel.augmentation.ApplyEachPolicy* attribute), 9

n\_per\_original (*snorkel.augmentation.MeanFieldPolicy* attribute), 10

n\_per\_original (*snorkel.augmentation.RandomPolicy* attribute), 12

name (*snorkel.classification.DictDataset* attribute), 20

name (*snorkel.classification.MultitaskClassifier* attribute), 25

name (*snorkel.classification.Operation* attribute), 29

name (*snorkel.classification.Task* attribute), 30

name (*snorkel.classification.Trainer* attribute), 32

name (*snorkel.labeling.LabelingFunction* attribute), 48

name (*snorkel.labeling.lf.nlp.NLPLabelingFunction* attribute), 53

name (*snorkel.labeling.lf.nlp\_spark.SparkNLPLabelingFunction* attribute), 59

name (*snorkel.slicing.sf.nlp.NLPSlicingFunction* attribute), 79

nlp\_labeling\_function (class in *snorkel.labeling.lf.nlp*), 61

nlp\_slicing\_function (class in *snorkel.slicing.sf.nlp*), 90

NLPLabelingFunction (class in *snorkel.labeling.lf.nlp*), 52

NLPSlicingFunction (class in *snorkel.slicing.sf.nlp*), 78

## O

op\_sequence (*snorkel.classification.Task* attribute), 30

op\_sequences (*snorkel.classification.MultitaskClassifier* attribute), 25

Operation (class in *snorkel.classification*), 29

optimizer (*snorkel.classification.Trainer* attribute), 32

output\_func (*snorkel.classification.Task* attribute), 30

output\_funcs (*snorkel.classification.MultitaskClassifier* attribute), 26

## P

PandasLFApplier (class in *snorkel.labeling*), 54

PandasParallelLFApplier (class in *snorkel.labeling.apply.dask*), 55

PandasParallelSFApplier (class in *snorkel.slicing.apply.dask*), 80

PandasSFApplier (class in *snorkel.slicing*), 80

PandasTFApplier (class in *snorkel.augmentation*), 11

predict() (*snorkel.classification.MultitaskClassifier* method), 28

predict() (*snorkel.labeling.model.baselines.MajorityClassVoter* method), 49

predict() (*snorkel.labeling.model.baselines.MajorityLabelVoter* method), 51

predict() (*snorkel.labeling.model.baselines.RandomVoter* method), 56

predict() (*snorkel.labeling.model.label\_model.LabelModel* method), 45

predict() (*snorkel.slicing.SliceAwareClassifier* method), 85

predict\_proba() (*snorkel.labeling.model.baselines.MajorityClassVoter* method), 49

predict\_proba() (*snorkel.labeling.model.baselines.MajorityLabelVoter* method), 51

predict\_proba() (*snorkel.labeling.model.baselines.RandomVoter* method), 57

predict\_proba() (*snorkel.labeling.model.label\_model.LabelModel* method), 46  
 preds\_to\_probs() (*in module snorkel.utils*), 94  
 Preprocessor (*class in snorkel.preprocess*), 72  
 preprocessor (*class in snorkel.preprocess*), 74  
 probs\_to\_preds() (*in module snorkel.utils*), 94

## R

RandomPolicy (*class in snorkel.augmentation*), 12  
 RandomVoter (*class in snorkel.labeling.model.baselines*), 56  
 reset() (*snorkel.classification.LogManager* method), 22  
 reset\_cache() (*snorkel.augmentation.TransformationFunction* method), 14  
 reset\_cache() (*snorkel.map.BaseMapper* method), 66  
 reset\_cache() (*snorkel.map.LambdaMapper* method), 67  
 reset\_cache() (*snorkel.map.Mapper* method), 68  
 reset\_cache() (*snorkel.preprocess.LambdaPreprocessor* method), 72  
 reset\_cache() (*snorkel.preprocess.nlp.SpacyPreprocessor* method), 74  
 reset\_cache() (*snorkel.preprocess.Preprocessor* method), 72  
 run() (*snorkel.augmentation.TransformationFunction* method), 14  
 run() (*snorkel.map.Mapper* method), 68  
 run() (*snorkel.preprocess.nlp.SpacyPreprocessor* method), 74  
 run() (*snorkel.preprocess.Preprocessor* method), 72  
 run\_log (*snorkel.classification.LogWriter* attribute), 23  
 run\_name (*snorkel.classification.LogWriter* attribute), 23  
 run\_name() (*snorkel.classification.LogWriterConfig* property), 25

## S

save() (*snorkel.classification.MultitaskClassifier* method), 28  
 save() (*snorkel.classification.Trainer* method), 33  
 save() (*snorkel.labeling.model.baselines.MajorityClassVoter* method), 50  
 save() (*snorkel.labeling.model.baselines.MajorityLabelVoter* method), 52  
 save() (*snorkel.labeling.model.baselines.RandomVoter* method), 57  
 save() (*snorkel.labeling.model.label\_model.LabelModel* method), 46  
 save() (*snorkel.slicing.SliceAwareClassifier* method), 85  
 score() (*snorkel.analysis.Scorer* method), 4  
 score() (*snorkel.classification.MultitaskClassifier* method), 28  
 score() (*snorkel.labeling.model.baselines.MajorityClassVoter* method), 50  
 score() (*snorkel.labeling.model.baselines.MajorityLabelVoter* method), 52  
 score() (*snorkel.labeling.model.baselines.RandomVoter* method), 57  
 score() (*snorkel.labeling.model.label\_model.LabelModel* method), 47  
 score() (*snorkel.slicing.SliceAwareClassifier* method), 85  
 score\_slices() (*snorkel.analysis.Scorer* method), 4  
 score\_slices() (*snorkel.slicing.SliceAwareClassifier* method), 86  
 Scorer (*class in snorkel.analysis*), 3  
 scorer (*snorkel.classification.Task* attribute), 30  
 scorers (*snorkel.classification.MultitaskClassifier* attribute), 26  
 seed (*snorkel.labeling.model.label\_model.LabelModel* attribute), 42  
 sequence\_length (*snorkel.augmentation.MeanFieldPolicy* attribute), 10  
 sequence\_length (*snorkel.augmentation.RandomPolicy* attribute), 12  
 SFApplier (*class in snorkel.slicing*), 81  
 slice\_dataframe() (*in module snorkel.slicing*), 90  
 slice\_ind\_key (*snorkel.slicing.SliceCombinerModule* attribute), 86  
 slice\_names (*snorkel.slicing.SliceAwareClassifier* attribute), 82  
 slice\_pred\_feat\_key (*snorkel.slicing.SliceCombinerModule* attribute), 87  
 slice\_pred\_key (*snorkel.slicing.SliceCombinerModule* attribute), 86  
 SliceAwareClassifier (*class in snorkel.slicing*), 82  
 SliceCombinerModule (*class in snorkel.slicing*), 86  
 slicing\_function (*class in snorkel.slicing*), 90  
 SlicingFunction (*class in snorkel.slicing*), 88  
 SpacyPreprocessor (*class in snorkel.preprocess.nlp*), 73  
 spark\_nlp\_labeling\_function (*class in snorkel.labeling.lf.nlp\_spark*), 62  
 SparkLFApplier (*class in snorkel.labeling.apply.spark*), 58  
 SparkNLPLabelingFunction (*class in snorkel.labeling.lf.nlp\_spark*), 58  
 SparkSFApplier (*in module snorkel.slicing.apply.spark*), 89  
 split (*snorkel.classification.DictDataset* attribute), 20

## T

Task (class in *snorkel.classification*), 30  
 task\_names (*snorkel.classification.MultitaskClassifier* attribute), 25  
 TensorBoardWriter (class in *snorkel.classification*), 31  
 TFAppplier (class in *snorkel.augmentation*), 13  
 to\_int\_label\_array() (in module *snorkel.utils*), 95  
 Trainer (class in *snorkel.classification*), 32  
 transformation\_function (class in *snorkel.augmentation*), 15  
 TransformationFunction (class in *snorkel.augmentation*), 14  
 trigger\_checkpointing() (*snorkel.classification.LogManager* method), 22  
 trigger\_evaluation() (*snorkel.classification.LogManager* method), 22

## U

update() (*snorkel.classification.LogManager* method), 22

## W

write\_config() (*snorkel.classification.LogWriter* method), 24  
 write\_config() (*snorkel.classification.TensorBoardWriter* method), 31  
 write\_json() (*snorkel.classification.LogWriter* method), 24  
 write\_json() (*snorkel.classification.TensorBoardWriter* method), 31  
 write\_log() (*snorkel.classification.LogWriter* method), 24  
 write\_log() (*snorkel.classification.TensorBoardWriter* method), 32  
 write\_text() (*snorkel.classification.LogWriter* method), 24  
 write\_text() (*snorkel.classification.TensorBoardWriter* method), 32  
 writer (*snorkel.classification.TensorBoardWriter* attribute), 31

## X

X\_dict (*snorkel.classification.DictDataset* attribute), 20

## Y

Y\_dict (*snorkel.classification.DictDataset* attribute), 20