SDK Reference v25.7.7

See all our docs at docs.snorkel.ai

PDF generated on March 4, 2026

Python SDK

The Snorkel AI Data Development Platform SDK lets you drive your development process programmatically.

- Read the SDK reference documentation.

Snorkel's Python SDK comes in two different distributions: the full SDK and the basic SDK.

- The full SDK is preinstalled on Snorkel's hosted in-platform Jupyter notebook server. Access it from **Notebook** in the left navigation.
- The basic SDK, a subset of the full SDK, can be installed in any Python environment. Read how to install the SDK locally in the SDK quickstart.

SDK quickstart

This guide shows you how to access or install Snorkel's SDK, and then use it to explore core data management functions.

- Install the SDK locally
- Access the SDK from a Snorkel-hosted instance

Install the SDK locally

Requirements

- Python version 3.9-3.11
- A Snorkel account with admin or developer access
- Your Snorkel API key
- Your Snorkel instance host name or IP address. You can copy the host name from your Snorkel instance's URL.

TIP

Use a Conda environment for your local Snorkel installation. For example, you can run these commands to create and activate a Python 3.11 environment:

```
conda create -n snorkel-env python=3.11

conda activate snorkel-env
```

Install the SDK

Run the following command to install the basic Snorkel SDK in your Python environment:

```
pip install 'snorkelai-sdk[default]' \
--index-url https://"{SNORKEL_PLATFORM_API_KEY}"@{your-snorkel-
hostname}/sdk/repo \
--extra-index-url https://pypi.org/simple
```

Known issues

- `--extra-index-url https://pypi.org/simple` must be referenced explicitly in the installation.

Authentication

All API requests to Snorkel must be authenticated. You need an API key, and the Python SDK client needs to use an authenticated `SnorkelSDKContext` object to make API requests to Snorkel services.

Follow these instructions to generate an API key:

- User and Admin settings

When you connect to Snorkel locally or from another external system, you must provide additional settings and authentication secrets. Use the following connection template:

```
import os

# Core Snorkel SDK imports
import snorkelai.sdk.client as sai
from snorkelai.sdk.develop import Dataset, Slice, Batch

print("✅ Snorkel AI SDK imported successfully")

# Snorkel SDK configuration
SAI_CONFIG = {
   "endpoint": "https://<your-snorkel-hostname>",
   "minio_endpoint": "https://<your-minio-endpoint>",
   "api_key": "<your-api-key>",
   "workspace_name": "<your-workspace-name>",
   "debug": True  # Optional: set to False to disable debug logging
}

# Initialize Snorkel context
ctx = sai.SnorkelSDKContext.from_endpoint_url(**SAI_CONFIG)
```

- `endpoint`: You can copy the host name from your Snorkel instance's URL.
- `api_key`: Your Snorkel API key
- Contact your Snorkel admin or support representative to obtain your `minio_endpoint`.

> ⓘ NOTE
>
> While this quickstart doesn't require MinIO access beyond the `minio_endpoint` configuration, you may need to set a `minio_access_key` and `minio_secret_key` for more advanced data operations. Contact your Snorkel representative for assistance.

Access the SDK from a Snorkel-hosted instance

Requirements

- A Snorkel account with admin or developer access

Access the SDK

You can call the Snorkel SDK from a Jupyter notebook hosted on your Snorkel instance.

1. Select **Notebook** from the left navigation.
2. Select **+** to create a new notebook from scratch, or upload one.
3. Select **Python 3 (ipykernel)** from the **Notebook** section. A new Jupyter notebook opens.

Authentication

When using a Snorkel-hosted notebook, Snorkel automatically generates an API key for your user and assigns it as an environment variable. When you log out and log back in, Snorkel rotates this key to ensure security.

```
import snorkelai.sdk.client as sai

# Set your workspace
workspace_name = 'YOUR_WORKSPACE_NAME' #INPUT - Replace with your workspace
name

# Configure client context for Snorkel instance
ctx = sai.SnorkelSDKContext.from_endpoint_url(
    workspace_name=workspace_name,
)
```

Note that the `workspace_name` is not required when using the default workspace.

Now you can use Snorkel's SDK.

Quickstart: SDK connection and dataset exploration

This quickstart guide will walk you through connecting to Snorkel and exploring your workspace using the SDK. You'll learn how to authenticate and explore core data management functions.

Getting started

Begin by importing the necessary packages and setting up your connection:

```
import snorkelai.sdk.client as sai
from snorkelai.sdk.develop import Dataset, Slice

# Connect to Snorkel using the authentication block from above
# ctx = sai.SnorkelSDKContext.from_endpoint_url( ... ) # Input external or
Snorkel-hosted connection details

print("✅ SDK imported successfully")
```

For the connection, uncomment the `ctx` line and replace it with the full authentication block from the appropriate section above.

Explore your workspace

Explore datasets available in your workspace:

```python
# Verify connection
print(f"Connected to workspace: {ctx.workspace_name}")

# List existing datasets in your workspace
# Note: For workspaces with many datasets, this may take a moment
print("\nRetrieving datasets from workspace...")
datasets = Dataset.list()
print(f"Found {len(datasets)} datasets in workspace")

# Show first few datasets only to avoid overwhelming output
max_display = 5
datasets_to_show = datasets[:max_display]
print(f"\nShowing first {len(datasets_to_show)} datasets:")

for dataset in datasets_to_show:
    print(f"- {dataset.name}")
    print(f"  UID: {dataset.uid}")
    print(f"  Created: {getattr(dataset, 'created_at', 'Unknown')}")
    print()

if len(datasets) > max_display:
    print(f"... and {len(datasets) - max_display} more datasets")
```

Create a dataset

Create a new dataset to demonstrate SDK capabilities:

```python
# Create a new dataset (metadata only - no data upload required)
import datetime
timestamp = datetime.datetime.now().strftime("%Y%m%d_%H%M%S")
dataset_name = f"quickstart-demo-{timestamp}"

new_dataset = Dataset.create(dataset_name)
print(f"✅ Created dataset: {new_dataset.name}")
print(f"Dataset UID: {new_dataset.uid}")
```

Display dataset metadata

Explore dataset information without accessing the underlying data:

```python
# Refresh dataset list to include our new dataset
datasets = Dataset.list()
print(f"\nNow have {len(datasets)} datasets in workspace")

# Show our newly created dataset and a couple others
print(f"\nRecent datasets (showing up to 3):")
recent_datasets = datasets[:3]  # Limit to first 3 for display
for dataset in recent_datasets:
    marker = "← NEW" if dataset.name == new_dataset.name else ""
    print(f"- {dataset.name} {marker}")
    print(f"  UID: {dataset.uid}")

if datasets:
    # Work with our newly created dataset
    dataset = new_dataset
    print(f"\nExploring our new dataset: {dataset.name}")
    print(f"Dataset UID: {dataset.uid}")

    # Show available dataset properties
    print(f"\nDataset properties:")
    for attr in ['name', 'uid', 'created_at', 'description']:
        if hasattr(dataset, attr):
            value = getattr(dataset, attr)
            print(f"- {attr}: {value}")

    print("✅ Dataset metadata exploration successful")
else:
    print("No datasets found.")
```

Explore dataset organization with slices

View how data is organized with slices:

```
if datasets:
    # Use the first existing dataset (not our empty new one)
    dataset = datasets[0] if datasets[0].name != new_dataset.name else
(datasets[1] if len(datasets) > 1 else datasets[0])

    # List existing slices for the dataset
    existing_slices = Slice.list(dataset=dataset.uid)
    print(f"\nSlices for dataset '{dataset.name}':")
    print(f"Found {len(existing_slices)} slices:")

    for slice_obj in existing_slices:
        print(f"- {slice_obj.name}")
        print(f"  Description: {slice_obj.description}")
        # Note: Some slice objects may not have uid attribute
        if hasattr(slice_obj, 'uid'):
            print(f"  UID: {slice_obj.uid}")
        print()

    print("✅ Slice exploration complete")
```

SDK method verification

Finally, verify key SDK functions:

```
print("\n=== SDK Method Verification ===")

# Test context properties
print(f"Workspace access: {'✅ Available' if hasattr(ctx, 'workspace_name')
else '❌ Missing'}")
print(f"Debug mode: {'✅ Available' if hasattr(ctx, 'set_debug') else '❌
Missing'}")

# Test Dataset class methods
print(f"Dataset listing: {'✅ Available' if hasattr(Dataset, 'list') else '❌
Missing'}")
print(f"Dataset creation: {'✅ Available' if hasattr(Dataset, 'create') else
'❌ Missing'}")

# Test Slice class methods
print(f"Slice listing: {'✅ Available' if hasattr(Slice, 'list') else '❌
Missing'}")
print(f"Slice creation: {'✅ Available' if hasattr(Slice, 'create') else '❌
Missing'}")

print(f"\nWorkspace summary:")
print(f"- Workspace: {ctx.workspace_name}")
print(f"- Total datasets: {len(datasets) if 'datasets' in locals() else 0}")
print(f"- SDK connection: ✅ Working")

print("\n✅ SDK exploration complete!")
```

In this tutorial, you've successfully connected to the Snorkel AI Data Development Platform and explored your workspace using the SDK. You've seen how to authenticate, list datasets, explore data organization, and verify SDK functionality. This provides the foundation for more advanced SDK workflows.

Next steps

Explore the SDK reference documentation.

Notebook tips

This document contains a collection of tips and best practices for using Jupyter notebooks effectively with Snorkel.

How to use Git CLI with Snorkel

Snorkel's in-platform notebook server comes with Git CLI, which allows users to version control notebooks, data files, and more.

1. Select **Notebook** from the left navigation.
2. From the **Other** menu, select **Terminal**.
3. Clone your git repository using the full HTTPS URL, like `https://github.com/USERNAME/REPO.git`:

   ```
   $ git clone https://github.com/USERNAME/REPO.git
   Username: YOUR_USERNAME
   Password: YOUR_PERSONAL_ACCESS_TOKEN
   ```

   Use your personal access token. See Using a personal access token on the command line for details.

> NOTE
>
> Note that SSH URLs, like `git@github.com:user/repo.git`, are not supported.

> NOTE
>
> Snorkel does not come with its own internal Git repository. Please create or provide your own from GitHub or equivalent.

The Git CLI is also available through the Notebook via system shell access (i.e., the magic command `!git`).

API Reference

`snorkelai.sdk.client`	Interfaces to interact with the Snorkel Flow REST API.
`snorkelai.sdk.utils`	Other general utilities.
`snorkelai.sdk.develop`	(Beta) Object-oriented SDK for Snorkel Flow

snorkelai.sdk.client

Interfaces to interact with the Snorkel Flow REST API.

Most of the functions in the `snorkelai.sdk.client` module require a *client context* object — `SnorkelSDKContext` — that points to the Snorkel Flow instance:

```
import snorkelai.sdk.client as sf
ctx = sf.SnorkelSDKContext.from_endpoint_url()
```

All the functions under submodules are also available under `snorkelai.sdk.client`.

Examples

```
import snorkelai.sdk.client as sf
# get_annotations is available under snorkelai.sdk.client.annotations
sf.annotations.get_annotations(node)
# also available under snorkelai.sdk.client (recommended)
sf.get_annotations(node)  # noqa: F405
```

Since `snorkelai.sdk.client` submodules may be reorganized in the future, we recommend accessing functions directly from `snorkelai.sdk.client` to minimize the risk of future breaking changes.

Submodules

`snorkelai.sdk.client.annotations`	Annotations allow subject matter experts (SMEs) and LLMs to add labels to your data.
`snorkelai.sdk.client.annotation_sources`	Annotation source related functions.
`snorkelai.sdk.client.comments`	Comment related functions.
`snorkelai.sdk.client.connector_configs`	Dataset views functions for datasets
`snorkelai.sdk.client.ctx`	Context related classes.
`snorkelai.sdk.client.external_models`	External model endpoints related functions.

`snorkelai.sdk.client.files`	File storage related functions to upload and download files and directories.
`snorkelai.sdk.client.file_storage_configs`	File storage config related functions.
`snorkelai.sdk.client.fm_suite`	Foundation model suite related functions.
`snorkelai.sdk.client.gts`	Ground truth related functions.
`snorkelai.sdk.client.secrets`	Secret store related functions.
`snorkelai.sdk.client.synthetic`	Synthetic data related functions for generating synthetic data.
`snorkelai.sdk.client.transfer`	SDK functions for transferring assets between applications or nodes on a single Snorkel Flow instance.
`snorkelai.sdk.client.utils`	Utility functions.
`snorkelai.sdk.client.users`	User related functions.

snorkelai.sdk.client.annotation_sources

Annotation source related functions.

Functions

`create_annotation_source`([username, ...])	Create an annotation source.
`delete_annotation_source`(source_name)	Delete an annotation source.
`get_annotation_source`([source_name, source_uid])	Get an annotation source from uid or name.
`get_annotation_sources`()	Get annotation sources.
`update_annotation_source`(source_name[, ...])	Update an annotation source.

snorkelai.sdk.client.annotation_sources.create_annotation_source

snorkelai.sdk.client.annotation_sources.create_annotation_source(*username=None, source_name=None, source_type=None, metadata=None*)

Create an annotation source.

Parameters

Name	Type	Default	Info
username	`Optional[str]`	`None`	The username of source if the source type is user.
source_name	`Optional[str]`	`None`	The name of the source, which defaults to username for user type of source.
source_type	`Optional[str]`	`None`	The type of source (user, aggregation, etc).
metadata	`Optional[dict]`	`None`	Any source metadata.

Returns

The created source.

Return type

`Source`

snorkelai.sdk.client.annotation_sources.delete_annotation_source

snorkelai.sdk.client.annotation_sources.delete_annotation_source(*source_name*)

Delete an annotation source.

Parameters

Name	Type	Default	Info
source_name	`str`		The name of the source.

Returns

Returns true if the operation succeeds.

Return type

`bool`

snorkelai.sdk.client.annotation_sources.get_annotation_source

snorkelai.sdk.client.annotation_sources.get_annotation_source(*source_name=None, source_uid=None*)

Get an annotation source from uid or name.

Parameters

Name	Type	Default	Info
source_name	`Optional[str]`	`None`	Name of the source (such as username, etc).
source_uid	`Optional[int]`	`None`	Uid of the source.

Return type

`A single source that matches name and/or uid`

snorkelai.sdk.client.annotation_sources.get_annotation_sources

snorkelai.sdk.client.annotation_sources.get_annotation_sources()

Get annotation sources.

Returns

A list of sources.

Return type

`List[Source]`

snorkelai.sdk.client.annotation_sources.update_annotation_source

snorkelai.sdk.client.annotation_sources.update_annotation_source(*source_name, new_source_name=None, metadata=None*)

Update an annotation source.

Parameters

Name	Type	Default	Info
source_name	`str`		The current name of the source.
new_source_name	`Optional[str]`	`None`	The new name of the source.
metadata	`Optional[dict]`	`None`	The updated metadata.

Returns

The updated source.

Return type

`Source`

snorkelai.sdk.client.annotations

Annotations allow subject matter experts (SMEs) and LLMs to add labels to your data. Collecting human labels as ground truth for a subset of your data is essential for calibrating the quality of generated labels. Annotations provide baselines for advanced automated data pipelines.

The methods in this module allow users to create, edit, and delete annotations, and calculate agreement metrics between different annotation sources. This module also provides methods for managing how annotations are registered to Snorkel as ground truth.

Functions

`add_annotation`(node, x_uid, label[, metadata])	Adds a single annotation to a document or span at the specified node in your data pipeline.
`add_annotations`(node, annotations[, ...])	Adds multiple annotations to a document or span at the specified node in your data pipeline, in a single batch operation, using a dictionary for the list.
`aggregate_annotations`(node, batch_name[, ...])	Combines annotations from multiple specified sources into a single aggregated set using the specified strategy.
`commit_annotations`(node, source_name)	Commits annotations from the specified source into ground truth labels that can be used for training and evaluation.
`delete_annotation`(node, annotation_uids)	Permanently deletes one or more annotations from the specified node in your data pipeline.
`get_annotations`(node[, source_name, x_uids, ...])	Gets annotations, with flexible filtering options, for datapoints from the specified node in your data pipeline.
`get_interannotator_agreement`(node[, ...])	Gets agreement statistics for annotations at the specified node in your data pipeline.
`update_annotation`(node, annotation_uid, label)	Updates the label or metadata for an existing annotation while preserving its original author and timestamp.

snorkelai.sdk.client.annotations.add_annotation

snorkelai.sdk.client.annotations.add_annotation(*node, x_uid, label, metadata=None*)

Adds a single annotation to a document or span at the specified node in your data pipeline.

Creates a new annotation with the specified label and optional metadata. The label format must match the expected format for the node's task type, such as classification or sequence tagging.

To learn more about label formats, see Format for ground truth interaction in the SDK.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where you want to add the annotation.
x_uid	`str`		The unique identifier of the document or span to annotate, such as `doc::1`.
label	`Any`		The annotation label. Format depends on the node's task type: • Classification: string, such as `"POS"` or `"NEG"`. • Multi-label: list of strings, such as `"PERSON"` or `"ORG"`. • Sequence: list of spans, including the `start`, `end`, and `label`, such as `[[0, 4, "COMPANY"]]`.
metadata	`Optional[Dict]`	`None`	Additional information to store with the annotation, such as `{"confidence": 0.9}`.

Raises

- **ValueError** – If the label format doesn’t match the node’s expected format.
- **ValueError** – If the `x_uid` doesn’t exist in the node.
- **ValueError** – If there are permission issues.

Return type

`Dict`

Examples

Example 1

Add a classification label:

```
# Add a classification label to doc::10001
classification_result = sf.add_annotation(
    node=123,
    x_uid="doc::10001",
    label="POS",
    metadata={"annotator": "ava.annotator", "confidence": 0.9}
)
print(classification_result)
```

Example 1 return

Sample output showing the annotation added to `doc::10001`:

```
{
    "annotation_uid": 9492734,
    "source": {
        "source_uid": 1920,
        "source_type": "user",
        "source_name": "ava.annotation",
        "user_uid": 2059,
        "metadata": {},
        "workspace_uid": 1
    }
}
```

Example 2

Add a sequence tagging label:

```python
# Add a sequence tagging label to doc::1001
sequence_result = sf.add_annotation(
    node=95051,
    x_uid="doc::1001",
    label=[[0, 4, "COMPANY"]]
)
print(sequence_result)
```

Example 2 return

Sample output showing the annotation added to `doc::1001`:

```
{
    "annotation_uid": 9492791,
    "source": {
        "source_uid": 1959,
        "source_type": "user",
        "source_name": "diego.developer",
        "user_uid": 2059,
        "metadata": {},
        "workspace_uid": 1
    }
}
```

snorkelai.sdk.client.annotations.add_annotations

snorkelai.sdk.client.annotations.add_annotations(*node, annotations, source_name=None, username=None*)

Adds multiple annotations to a document or span at the specified node in your data pipeline, in a single batch operation, using a dictionary for the list.

Provides bulk annotation creation, which is more efficient than `add_annotation()`. Each annotation must specify an `x_uid` (document/span identifier), a label, and a shared source. The label must align with the configuration of the model node.

You must specify either a `source_name` or a `username` to associate with the annotations (but not both):

- Use `source_name` for non-user annotation sources like `majority_vote` or `external_import`.
- Use `username` for annotations from a specific user. If the username doesn't exist, this function creates a new `source_name`.

To learn more about label formats, see Format for ground truth interaction in the SDK.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where you want to add the annotations.
annotations	`List[Dict[str, Any]]`		List of annotation dictionaries. Each must contain: • `x_uid`: `str` - The unique identifier of the document or span to annotate, such as `doc::1`. • `label`: `Any` - The annotation label. Format

			depends on the node's task type: ◦ Classification: string, such as `"POS"` or `"NEG"` ◦ Multi-label: list of strings, such as `["PERSON", "ORG"]` ◦ Sequence: list of spans, including the `start`, `end`, and `label`, such as `[[0, 4, "COMPANY"]]` Optional fields: • `metadata`: `Dict` - Additional information to store with the annotation, such as `{"confidence": 0.9}` • `ts`: `str` - Timestamp in ISO format, such as `2023-01-01T12:00:00`
source_name	`Optional[str]`	`None`	Name of a non-user annotation source, such as `majority_vote` or `external_import`. Cannot be used together with `username`.
username	`Optional[str]`	`None`	Username to associate with the annotations. If username doesn't exist, creates a new source. Cannot be used together with `source_name`.

Raises

- **ValueError** – If both `source_name` and `username` are provided or neither is provided.
- **ValueError** – If any annotation is missing required fields.
- **ValueError** – If the provided label format is invalid for the node type.
- **ValueError** – If `source_uid` is manually specified in annotations.
- **ValueError** – If the operation fails.

Return type

`List[Dict[str, Any]]`

Examples

Example 1

Add multiple classification labels from a specific user:

```
annotations_to_add = [
    {"x_uid": "doc::20008", "label": "stock", "metadata": {"confidence":
0.9}},
    {"x_uid": "doc::20009", "label": "employment", "metadata":
{"confidence": 0.8}},
    {"x_uid": "doc::20010", "label": "loan", "metadata": {"confidence":
0.9}}
]

# Add multiple classification labels from a specific user, ava.annotator
bulk_classification_results = sf.add_annotations(
    node=123,
    annotations=annotations_to_add,
    username="ava.annotator"
)

print(bulk_classification_results)
```

Example 1 return

Sample output showing the annotations added to `doc::20008`, `doc::20009`, and `doc::20010`:

```
[
    {
        "annotation_uid": 9492793,
        "x_uid": "doc::20008",
        "source_uid": 1952
    },
    {
        "annotation_uid": 9492752,
        "x_uid": "doc::20009",
        "source_uid": 1952
    },
    {
        "annotation_uid": 9492753,
        "x_uid": "doc::20010",
        "source_uid": 195
    }
]
```

Example 2

Add multiple sequence tagging labels from the non-user source, external_import:

```
annotations_to_add = [
    {
        "x_uid": "doc::1010",
        "label": [[0, 4, "COMPANY"]],
        "metadata": {"confidence": 0.9}
    },
    {
        "x_uid": "doc::1012",
        "label": [[5, 12, "COMPANY"]],
        "metadata": {"confidence": 0.85}
    },
    {
        "x_uid": "doc::1017",
        "label": [[0, 7, "OTHER"]],
        "metadata": {"annotator_notes": "Remove company label"}
    }
]

# Add multiple sequence tagging labels from the non-user source,
external_import
bulk_sequence_results = sf.add_annotations(
    node=95051,
    annotations=annotations_to_add,
    source_name="external_import"
)

print(bulk_sequence_results)
```

Example 2 return

Sample output showing the annotations added to `doc::1010`, `doc::1012`, and `doc::1017`:

```
[
    {
        "annotation_uid": 9492796,
        "x_uid": "doc::1010",
        "source_uid": 1954
    },
    {
        "annotation_uid": 9492797,
        "x_uid": "doc::1012",
        "source_uid": 1954
    },
    {
        "annotation_uid": 9492798,
        "x_uid": "doc::1017",
        "source_uid": 1954
    }
]
```

snorkelai.sdk.client.annotations.aggregate_annotations

snorkelai.sdk.client.annotations.aggregate_annotations(*node, batch_name, sources=None, strategy=None*)

Combines annotations from multiple specified sources into a single aggregated set using the specified strategy.

This set of annotations is visible as a new batch on the **Batches** page in the GUI.

Note: Resample the data splits to ensure you can see the aggregated annotations.

The aggregation strategy depends on the node's task type, and each of these strategies handles conflicts differently:

- Multi-class classification: `simple_majority` selects most frequent label.
- Multi-label classification: `simple_union` takes union of all votes, breaking ties by frequency.
- Sequence tagging: `simple_intersection` selects spans marked by all annotators.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where you want to aggregate the annotations.
batch_name	`str`		Name for the batch of aggregated annotations.
sources	`Optional[List[str]]`	`None`	List of source names to aggregate. If `None`, uses all available sources.
strategy	`Optional[str]`	`None`	Aggregation strategy to use. If `None`, uses default for task type:

			• `simple_majority` for classification. • `simple_union` for multi-label. • `simple_intersection` for sequence tagging.

Raises

- **ValueError** – If the batch name doesn’t exist.
- **ValueError** – If any source name in `sources` doesn’t exist.
- **ValueError** – If the aggregation operation fails.
- **ValueError** – If the source UID doesn’t exist.
- **ValueError** – If there are permission issues.
- **RuntimeError** – If no valid API key is provided.

Return type

`List`[`Annotation`]

Examples

Example 1

Aggregate annotations from multiple sources:

```
# Aggregate annotations from two annotators using simple_majority
strategy
aggregated_annotations = sf.aggregate_annotations(
    node=123,
    batch_name="consensus_batch",
    sources=["ava.annotator", "rosa.reviewer"],
    strategy="simple_majority"
)
print(aggregated_annotations)
```

Example 1 return

Sample output showing the aggregated annotations:

```
__DATAPOINT_UID  annotation_uid
doc::10005       9492734           {'annotation_uid': 9492734, 'x_uid':
'doc::10005',
                                  'source_uid': 1952, 'label': 'POS',
                                  'metadata': {'aggregation_strategy':
'simple_majority'},
                                  'ts': datetime.datetime(2025, 2, 19,
17, 7, 36)}
doc::20001       9492743           {'annotation_uid': 9492743, 'x_uid':
'doc::20001',
                                  'source_uid': 1952, 'label': 'NEG',
                                  'metadata': {'aggregation_strategy':
'simple_majority'},
                                  'ts': datetime.datetime(2025, 2, 19,
17, 7, 36)}
doc::20003       9492744           {'annotation_uid': 9492744, 'x_uid':
'doc::20003',
                                  'source_uid': 1952, 'label': 'POS',
                                  'metadata': {'aggregation_strategy':
'simple_majority'},
                                  'ts': datetime.datetime(2025, 2, 19,
17, 7, 36)}
```

snorkelai.sdk.client.annotations.commit_annotations

snorkelai.sdk.client.annotations.commit_annotations(*node*, *source_name*)

Commits annotations from the specified source into ground truth labels that can be used for training and evaluation. After collecting annotations from your subject matter experts or an LLM, you can commit the annotations as ground truth labels.

Note: This function may change labeling function and model metrics.

Note: The commit operation is permanent for the current version.

Note: To see the new ground truth, you might need to resample the data split.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where the annotations exist.
source_name	`str`		Name of the annotation source to commit, such as `annotator_name` or `majority_vote`. This can be a `username` or a `source_name` used with the `aggregate_annotations` function.

Raises

- **ValueError** – If the `source_name` doesn’t exist.
- **ValueError** – If there are no annotations for the source.
- **ValueError** – If there are permission issues.
- **ValueError** – If the commit operation fails.

Return type

`None`

Examples

Example 1

Commit annotations from a specified annotator as ground truth:

```
# Commit annotations from a ava.annotator as ground truth
sf.commit_annotations(
    node=123,
    source_name="ava.annotator"
)
```

If the function does not raise an error, the annotations were committed successfully.

Example 2

After aggregating annotations, commit the consensus version of the labels as ground truth:

```
# Commit annotations from the consensus_batch source_name as ground
truth
sf.commit_annotations(
    node=123,
    source_name="consensus_batch"
)
```

If the function does not raise an error, the annotations were committed successfully.

snorkelai.sdk.client.annotations.delete_annotation

snorkelai.sdk.client.annotations.delete_annotation(*node*, *annotation_uids*)

Permanently deletes one or more annotations from the specified node in your data pipeline.

Deleted annotations cannot be recovered. Before deleting an annotation, use `get_annotations()` to see all annotations for a node and to retrieve the appropriate annotation UIDs.

Deleting an annotation does not affect ground truth labels, even if the annotation was previously committed as ground truth.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where the annotations exist.
annotation_uids	`List[int]`		List of annotation UIDs to delete. Get these from `get_annotations()`.

Raises

- **ValueError** – If any `annotation_uid` doesn’t exist
- **ValueError** – If there are permission issues
- **ValueError** – If the deletion operation fails

Return type

`None`

Examples

Example 1

Delete a single annotation:

```
# Delete annotation 456 from node 123
sf.delete_annotation(
    node=123,
    annotation_uids=[456]
)
```

If the function does not raise an error, the annotation was deleted successfully.

Example 2

Delete multiple annotations:

```
# Delete annotations 456, 789, and 1011 from node 123
sf.delete_annotation(
    node=123,
    annotation_uids=[456, 789, 1011]
)
```

If the function does not raise an error, the annotations were deleted successfully.

snorkelai.sdk.client.annotations.get_annotations

snorkelai.sdk.client.annotations.get_annotations(*node, source_name=None, x_uids=None, batch_uids=None*)

Gets annotations, with flexible filtering options, for datapoints from the specified node in your data pipeline.

Gets annotations as a Pandas Series, indexed by (`x_uid`, `annotation_uid`) where the datapoint ID is `x_uid` and the annotation ID is `annotation_uid`. You can filter annotations by specifying:

- A specific annotator or source using `source_name`.
- Specific datapoints using `x_uids`.
- Specific batches using `batch_uids`, which can be accessed by using the get_batches() function.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node containing the annotations.
source_name	`Optional[str]`	`None`	Filters annotations to include only those from a specific annotator or source, such as `annotator_name` or `majority_vote`. Can be used together with `x_uids` but not with `batch_uids`.
x_uids	`Optional[List[str]]`	`None`	Filters annotations to include only those from specific datapoint IDs, such as `["doc::1", "doc::2"]`. Can be used together with

			`source_name` but not with `batch_uids`.
batch_uids	`Optional[List[int]]`	`None`	Filters annotations to include only those from specific batch IDs. Get batch IDs using get_batches(). Cannot be used together with `source_name` or `x_uids`.

Raises

ValueError – If `batch_uids` and (`source_name`, `x_uids`) are provided simultaneously.

Return type

`Series`

Examples

Example 1

Get all annotations for a node:

```
# Get all annotations for node 123
annotations = sf.get_annotations(node=123)
print(annotations)
```

Example 1 return

Sample output showing annotations for all documents from node 123:

```
__DATAPOINT_UID  annotation_uid
doc::10005       9492734          {'annotation_uid': 9492734, 'x_uid':
'doc::100...
                 9492781          {'annotation_uid': 9492781, 'x_uid':
'doc::100...
doc::20001       9492743          {'annotation_uid': 9492743, 'x_uid':
'doc::200...
                 9492754          {'annotation_uid': 9492754, 'x_uid':
'doc::200...
                 9492767          {'annotation_uid': 9492767, 'x_uid':
'doc::200...

...
```

Example 2

Get annotations for specific documents:

```
# Get annotations for doc::20001 and doc::20002
annotations = sf.get_annotations(
    node=123,
    x_uids=["doc::20001", "doc::20002"]
)
print(annotations)
```

Example 2 return

Sample output showing annotations for specific documents:

```
__DATAPOINT_UID  annotation_uid
doc::20001       9492743          {'annotation_uid': 9492743, 'x_uid':
'doc::200...
                 9492754          {'annotation_uid': 9492754, 'x_uid':
'doc::200...
                 9492767          {'annotation_uid': 9492767, 'x_uid':
'doc::200...
doc::20002       9492765          {'annotation_uid': 9492765, 'x_uid':
'doc::200...
                 9492779          {'annotation_uid': 9492779, 'x_uid':
'doc::200...
```

Example 3

Get annotations from a specific annotator:

```
# Get annotations from annotator ava.annotator
annotations = sf.get_annotations(
    node=123,
    source_name="ava.annotator"
)
print(annotations)
```

Example 3 return

Sample output showing annotations for a specific annotator:

```
__DATAPOINT_UID  annotation_uid
doc::20001       9492743          {'annotation_uid': 9492743, 'x_uid':
'doc::200...
doc::20003       9492744          {'annotation_uid': 9492744, 'x_uid':
'doc::200...
doc::20004       9492745          {'annotation_uid': 9492745, 'x_uid':
'doc::200...
doc::20006       9492746          {'annotation_uid': 9492746, 'x_uid':
'doc::200...
doc::20007       9492747          {'annotation_uid': 9492747, 'x_uid':
'doc::200...
doc::20008       9492783          {'annotation_uid': 9492783, 'x_uid':
'doc::200...
doc::20009       9492752          {'annotation_uid': 9492752, 'x_uid':
'doc::200...
doc::20010       9492753          {'annotation_uid': 9492753, 'x_uid':
'doc::200...
doc::20021       9492761          {'annotation_uid': 9492761, 'x_uid':
'doc::200...
doc::20023       9492762          {'annotation_uid': 9492762, 'x_uid':
'doc::200...
doc::20025       9492763          {'annotation_uid': 9492763, 'x_uid':
'doc::200...
```

Example 4

Get annotations for specific batches:

```
# Get annotations for batch 456
annotations = sf.get_annotations(
    node=123,
    batch_uids=[456]
)
print(annotations)
```

Example 4 return

Sample output showing annotations for a specific batch:

```
__DATAPOINT_UID  annotation_uid
doc::20008       9492783          {'annotation_uid': 9492783, 'x_uid':
'doc::200...
doc::20012       9492760          {'annotation_uid': 9492760, 'x_uid':
'doc::200...
                 9492777          {'annotation_uid': 9492777, 'x_uid':
'doc::200...
```

snorkelai.sdk.client.annotations.get_interannotator_agreement

snorkelai.sdk.client.annotations.get_interannotator_agreement(*node*, *label_str=None*, *batch_uids=None*, *metric=None*)

Gets agreement statistics for annotations at the specified node in your data pipeline.

This function helps you measure the quality and consistency of your annotations by calculating how well your annotators agree with each other, whether the annotator is a human or an LLM. It's useful for identifying potential issues in your annotation process or areas where annotators or an LLM-as-a-judge might need additional training.

The function computes pairwise agreement scores between annotators and an overall agreement metric. Agreement is calculated only for datapoints where multiple annotators provided labels.

Returns a symmetric agreement matrix where:

- Diagonal values are 1.0 (self-agreement).
- Off-diagonal values show pairwise agreement between annotators.
- Only considers datapoints with overlapping annotations.

Task-specific metrics:

- Classification/Multi-label: Krippendorff's Alpha.
- Sequence tagging: Mean Span F1.

Because different metrics are appropriate for different task types, do a batch-specific analysis to identify problematic batches. For example, if you have a classification task, you might want to calculate agreement for each class label separately.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where the annotations exist.

label_str	`Optional[str]`	`None`	If provided, calculate agreement only for this specific class label.
batch_uids	`Optional[List[int]]`	`None`	If provided, calculate agreement only for these specific batches.
metric	`Optional[str]`	`None`	Agreement metric to use. Available options are `krippendorff-alpha` for classification tasks and `mean-span-f1` for sequence tagging tasks.

Raises

- **ValueError** – If `label_str` is provided but not found in the node’s label map.
- **ValueError** – If there are insufficient overlapping annotations to calculate agreement.
- **ValueError** – If an invalid metric is specified for the task type.
- **RuntimeError** – If no valid API key is provided.

Return type

`Dict[str, Any]`

Examples

Example 1

Get overall agreement statistics for all annotators:

```
# Get overall agreement statistics for all annotators
overall_agreement = sf.get_interannotator_agreement(node=123)
print(overall_agreement)
```

Example 1 return

Sample output showing the agreement matrix for all annotators:

```
{
    "usernames": ["alex.annotator", "ava.annotator", "rosa.reviewer"],
    "matrix": [[1.0, null, null], [null, 1.0, 1.0], [null, 1.0, 1.0]],
    "metric": null
}
```

Example 2

Get agreement for a specific class label:

```
# Get agreement for a specific class label, loan
class_agreement = sf.get_interannotator_agreement(
    node=123,
    label_str="loan",
    metric="krippendorff-alpha"
)
print(class_agreement)
```

Example 2 return

Sample output showing the agreement for the `loan` class label:

```
{
    "usernames": ["alex.annotator", "ava.annotator", "rosa.reviewer"],
    "matrix": [[1.0, null, null], [null, 1.0, 1.0], [null, 1.0, 1.0]],
    "metric": -0.2592592592592593
}
```

snorkelai.sdk.client.annotations.update_annotation

snorkelai.sdk.client.annotations.update_annotation(*node, annotation_uid, label, metadata=None*)

Updates the label or metadata for an existing annotation while preserving its original author and timestamp.

This function modifies an existing annotation rather than creating a new one. For a complete overwrite (including author and timestamp), use delete_annotation() followed by add_annotation() instead.

Parameters

Name	Type	Default	Info
node	`int`		The unique identifier (UID) of the node in your data pipeline where the annotation exists.
annotation_uid	`int`		The unique identifier of the annotation to update. Get this from get_annotations().
label	`str`		The new label value. Must match the node’s expected label format.
metadata	`Optional[Dict]`	`None`	New metadata to associate with the annotation. If provided, completely replaces existing metadata.

Raises

- **ValueError** – If the `annotation_uid` doesn’t exist.
- **ValueError** – If the label format is invalid for the specified node.
- **ValueError** – If there are permission issues.
- **ValueError** – If the node information cannot be fetched.

Return type

`None`

Examples

Example 1

Here is an original annotation with a classification label:

```
sf.get_annotations(123).loc["doc::1009"].loc[456]
```

This returns:

```
{
    "annotation_uid": 456,
    "x_uid": "doc::1009",
    "label": "loan"
}
```

Update the classification label:

```
# Update the label
sf.update_annotation(
    node=123,
    annotation_uid=456,
    label="stock",
    metadata={"updated_by": "rosa.reviewer", "reason": "corrected after
review"}
)
```

Confirm your annotation is updated:

```
# Check updated annotation
sf.get_annotations(123).loc["doc::1009"].loc[456]
```

Example 1 return

Sample output showing the updated annotation:

```
{
    "annotation_uid": 9492734,
    "x_uid": "doc::10005",
    "source_uid": 1920,
    "label": "loan",
    "metadata": {
        "annotator": "ava.annotator",
        "confidence": 0.9
    },
    "ts": "2025-02-19T17:07:36"
}
```

Example 2

Here is an original annotation with a sequence tagging label:

```
sf.get_annotations(95051).loc["doc::1008"].loc[9492791]
```

This returns:

```
{
    "annotation_uid": 9492791,
    "x_uid": "doc::1008",
    "source_uid": 1959,
    "label": [[0, 4, "COMPANY"]],
    "metadata": {},
    "ts": "2025-03-26T15:54:02"
}
```

Update the sequence tagging annotation:

```
# Update the label
sf.update_annotation(
    node=95051,
    annotation_uid=9492791,
    label=[[0, 4, "OTHER"]],
    metadata={"reason": "remove company label"}
)
```

Confirm your annotation is updated:

```
# Check updated annotation
print(sf.get_annotations(95051).index.levels[0])
```

Example 2 return

Sample output showing the updated annotation:

```
{
    "annotation_uid": 9492791,
    "x_uid": "doc::1008",
    "source_uid": 1959,
    "label": [[0, 4, "OTHER"]],
    "metadata": {"reason": "increase span"},
    "ts": "2025-03-26T15:54:02"
}
```

snorkelai.sdk.client.comments

Comment related functions. Comments let you add custom notes to data points, for tracking and collaboration. Comments are visible to all users who have access to the task.

Functions

`create_comment`(node, x_uid, username, body)	Create a new comment for a data point.
`delete_comment`(node, comment_uid)	Remove an existing comment by its comment UID.
`delete_datapoint_comments`(node, x_uid)	Removes all existing comments for a particular datapoint.
`edit_comment`(node, comment_uid, body)	Edit the body text of an existing comment.
`get_comment`(node, comment_uid)	Fetch an existing comment by its UID.
`get_comments`(node[, username, x_uid])	Return a list of all comments for the task.

snorkelai.sdk.client.comments.create_comment

snorkelai.sdk.client.comments.create_comment(*node, x_uid, username, body, is_context_tag=False*)

Create a new comment for a data point.

Examples

```
>>> sf.create_comment(node=1, x_uid="doc::10005", username="user1",
body="This is a comment")
{
    'comment_uid': 7,
    'user_uid': 3,
    'x_uid': 'doc::10005',
    'body': 'This is a comment',
    'created_at': '2023-09-26T17:31:15.759565',
    'is_edited': False
}
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the parent model node for this comment.
x_uid	`str`		The UID for the datapoint. Datapoint UIDs can be found by looking at the index `__DATAPOINT_UID` column of the node's DataFrame.
username	`str`		The username of the author of the comment. A username must be provided.
body	`str`		The body text of the comment.
is_context_tag	`bool`	`False`	For information extraction tasks, adds a comment to the parent document instead of the provided span UID. By default False.

Returns

A dictionary of metadata corresponding to the newly created comment.

Return type

`Dict[str, Union[str, int]]`

snorkelai.sdk.client.comments.delete_comment

snorkelai.sdk.client.comments.delete_comment(*node, comment_uid*)

Remove an existing comment by its comment UID.

To delete all comments for a given datapoint, see delete_datapoint_comments()

Examples

```
>>> sf.delete_comment(node=1, comment_uid=7)
None
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the parent model node for this comment.
comment_uid	`int`		UID of the comment to delete.

Return type

`None`

snorkelai.sdk.client.comments.delete_datapoint_comments

snorkelai.sdk.client.comments.delete_datapoint_comments(*node*, *x_uid*)

Removes all existing comments for a particular datapoint.

To delete an individual comment by comment_uid, see delete_comment()

Examples

```
>>> sf.delete_datapoint_comments(node=1, x_uid="doc::10005")
None
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the parent model node for this comment.
x_uid	`str`		The UID of the datapoint whose comments we wish to delete.

Return type

`None`

snorkelai.sdk.client.comments.edit_comment

snorkelai.sdk.client.comments.edit_comment(*node, comment_uid, body*)

Edit the body text of an existing comment. This will overwrite the existing comment body. This will not change the comment's UID, creation date, or the associated username.

Examples

```
>>> sf.edit_comment(node=1, comment_uid=7, body="This is an edited
comment")
{
    'comment_uid': 7,
    'user_uid': 3,
    'x_uid': 'doc::10005',
    'body': 'This is an edited comment',
    'created_at': '2023-09-26T17:31:15.759565',
    'is_edited': True
}
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the parent model node for this comment.
comment_uid	`int`		UID of the comment to edit.
body	`str`		New body of the comment.

Returns

A dictionary of metadata corresponding to the edited comment.

Return type

`Dict[str, Union[str, int]]`

snorkelai.sdk.client.comments.get_comment

snorkelai.sdk.client.comments.get_comment(*node*, *comment_uid*)

Fetch an existing comment by its UID.

Examples

```
>>> sf.get_comment(node=1, comment_uid=7)
{
    'comment_uid': 7,
    'user_uid': 3,
    'x_uid': 'doc::10005',
    'body': 'This is a comment',
    'created_at': '2023-09-26T17:31:15.759565',
    'is_edited': False
}
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the parent model node for this comment.
comment_uid	`int`		UID of the comment to retrieve. Use `snorkelai.sdk.client.get_comments()` to see a list of all comments for a node.

Returns

A dictionary of metadata corresponding to the comment.

Return type

`Dict[str, Union[str, int]]`

snorkelai.sdk.client.comments.get_comments

snorkelai.sdk.client.comments.get_comments(*node, username=None, x_uid=None*)

Return a list of all comments for the task. This method can optionally be filtered by the username of the comment author or by the UID of the associated datapoint.

Examples

```
>>> sf.get_comments(node=1, username="user1")
[
    {
        'comment_uid': 7,
        'user_uid': 3,
        'x_uid': 'doc::10005',
        'body': 'This is a comment',
        'created_at': '2023-09-26T17:31:15.759565',
        'is_edited': False
    },
    {
        'comment_uid': 8,
        'user_uid': 3,
        'x_uid': 'doc::10005',
        'body': 'This is another comment',
        'created_at': '2023-09-26T17:31:15.759565',
        'is_edited': False
    }
]
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the parent model node for this comment.
username	`Optional[str]`	`None`	If included, only return comments by the user with this username.
x_uid	`Optional[str]`	`None`	UID of datapoint. If included, returns all comments associated with only this datapoint.

Returns

A list of dictionaries of metadata for all comments that match the provided filters.

Return type

`List[Dict[str, Union[str, int]]]`

snorkelai.sdk.client.connector_configs

Dataset views functions for datasets

Functions

`create_connector_config`(name, ...)	Create a new connector configuration.
`delete_connector_config`(config_uid)	Delete a connector configuration.
`get_connector_config`(config_uid)	Get a connector configuration by its UID.
`list_connector_configs`(connector_type)	List all configurations for a specific connector type.
`update_connector_config`(config_uid, ...)	Update a connector configuration.

snorkelai.sdk.client.connector_configs.create_connector_config

snorkelai.sdk.client.connector_configs.create_connector_config(*name, connector_type, config*)

Create a new connector configuration.

Parameters

Name	Type	Default	Info
name	`str`		The name of the connector configuration.
connector_type	`str`		The type of connector to create a configuration (e.g. “AmazonS3”).
config	`Dict[str, Any]`		The connector configuration.

Returns

The UID of the created connector configuration

Return type

`int`

Examples

```
>>> from snorkelai.sdk.client import create_connector_config
>>> config_uid = create_connector_config(
...     name="my_s3_config",
...     connector_type="AmazonS3",
...     config={
...         "access_key_id": "my_access_key_id",
...         "secret_access_key": "my_secret_access_key",
...         "region": "us-east-1",
...     },
... )
>>> print(config_uid)
```

snorkelai.sdk.client.connector_configs.delete_connector_config

snorkelai.sdk.client.connector_configs.delete_connector_config(*config_uid*)

Delete a connector configuration.

Parameters

Name	Type	Default	Info
config_uid	`int`		The UID of the connector configuration to delete.

Return type

`None`

Examples

```
>>> from snorkelai.sdk.client import delete_connector_config
>>> delete_connector_config(123)
```

snorkelai.sdk.client.connector_configs.get_connector_config

snorkelai.sdk.client.connector_configs.get_connector_config(*config_uid*)

Get a connector configuration by its UID.

Parameters

Name	Type	Default	Info
config_uid	`int`		The UID of the connector configuration to get.

Returns

A dictionary representing the connector configuration

Return type

`Dict[str, Any]`

Examples

```
>>> from snorkelai.sdk.client import get_connector_config
>>> config = get_connector_config(123)
>>> print(config)
```

snorkelai.sdk.client.connector_configs.list_connector_configs

snorkelai.sdk.client.connector_configs.list_connector_configs(*connector_type*)

List all configurations for a specific connector type.

Parameters

Name	Type	Default	Info
connector_type	`str`		The type of connector to list configurations for (e.g. “AmazonS3”).

Returns

A list of connector configurations of the given type

Return type

`List[Dict[str, Any]]`

Examples

```
>>> from snorkelai.sdk.client import list_connector_configs
>>> configs = list_connector_configs("AmazonS3")
>>> print(configs)
```

snorkelai.sdk.client.connector_configs.update_connector_config

snorkelai.sdk.client.connector_configs.update_connector_config(*config_uid, new_name, new_config*)

Update a connector configuration.

Parameters

Name	Type	Default	Info
config_uid	`int`		The UID of the connector configuration to update.
new_name	`str`		The new name of the connector configuration.
new_config	`Dict[str, Any]`		The new configuration for the connector.

Return type

`None`

Examples

```
>>> from snorkelai.sdk.client import update_connector_config
>>> new_config = {
...     "access_key_id": "new_access_key_id",
...     "secret_access_key": "new_secret_access_key",
...     "region": "new_region",
... }
>>> update_connector_config(123, "new_name", new_config)
```

snorkelai.sdk.client.ctx.SnorkelSDKContext

class snorkelai.sdk.client.ctx.SnorkelSDKContext(*tdm_client, minio_url=None, storage_client=None, workspace_name=None, set_global=True, debug=False*)

Bases: `object`

The SnorkelSDKContext object provides client context for the Snorkel Flow SDK. It allows the Snorkel Flow SDK to recognize a Snorkel Flow instance by identifying Snorkel Flow's essential API services (TDM, Studio, MinIO) via user-provided parameters, a YAML config file, or a Snorkel Flow API key.

Examples

```
import snorkelai.sdk.client as sf
ctx = sf.SnorkelSDKContext.from_endpoint_url(...)
```

__init__

__init__(*tdm_client, minio_url=None, storage_client=None, workspace_name=None, set_global=True, debug=False*)

Initialize a SnorkelSDKContext.

Methods

`__init__`(tdm_client[, minio_url, ...])	Initialize a SnorkelSDKContext.
`from_endpoint_url`([endpoint, ...])	Initialize a SnorkelSDKContext from keyword arguments.
`get_global`()	Retrieve the global SnorkelSDKContext object.
`set_debug`(debug)	Set the verbosity of warnings, details, and stacktraces
`set_global`([ctx])	Set a SnorkelSDKContext object globally.

Attributes

`storage_client`	

`workspace_name`	SnorkelSDKContext objects are only scoped to work in a particular workspace.

from_endpoint_url

classmethod **from_endpoint_url**(*endpoint=None, minio_endpoint=None, api_key=None, workspace_name=None, set_global=True, debug=False, *args, **kwargs*)

Initialize a SnorkelSDKContext from keyword arguments.

Examples

```
# Instantiate with default kwargs and a custom url
import snorkelai.sdk.client as sf
ctx =
sf.SnorkelSDKContext.from_endpoint_url("https://edge.k8s.g498.io/")
```

```
# Instantiate with custom kwargs
import snorkelai.sdk.client as sf
ctx = sf.SnorkelSDKContext.from_endpoint_url(
    endpoint="https://edge.k8s.g498.io/",
    minio_endpoint="https://edge-minio-api.k8s.g498.io",
    workspace_name="my-workspace",
)
```

```
# Instantiate with an API key
import snorkelai.sdk.client as sf
ctx = sf.SnorkelSDKContext.from_endpoint_url(
    endpoint="https://edge.k8s.g498.io/",
    api_key="my-api-key",
)
```

Parameters

Name	Type	Default	Info
endpoint	`Optional[str]`	`None`	The baseurl to a snorkelflow instance.
minio_endpoint	`Optional[str]`	`None`	The url to a minio service.

workspace_name	`Optional[str]`	`None`	The workspace name, which determines the workspace a dataset and application will be created in and queried from.
api_key	`Optional[str]`	`None`	The API key to use for the TDM, Studio, and Storage APIs.
set_global	`bool`	`True`	Whether to set the context as the global context, accessible across all notebooks in-platform.
debug	`bool`	`False`	Whether to enable debug mode. Debug mode will print more verbose errors to the screen inside of the Python notebook.

Returns

A SnorkelSDKContext object that can be used to interact with the MinIO, TDM, and Studio clients directly.

Return type

SnorkelSDKContext

get_global

classmethod **get_global**()

Retrieve the global SnorkelSDKContext object.

Examples

```
import snorkelai.sdk.client as sf
ctx = sf.SnorkelSDKContext.get_global()
```

Returns

A SnorkelSDKContext object that can be used to interact with the MinIO, TDM, and Studio clients directly.

Return type

SnorkelSDKContext

Raises

AttributeError – If no global context has been set.

set_debug

set_debug(*debug*)

Set the verbosity of warnings, details, and stacktraces

Parameters

Name	Type	Default	Info
debug	`bool`		If True, SDK operations will print more verbose errors to the screen inside of the Python notebook.

Return type

`None`

set_global

classmethod **set_global**(*ctx=None*)

Set a SnorkelSDKContext object globally. This context object will be used by all Snorkel Platform SDK functions.

Examples

```
import snorkelai.sdk.client as sf
ctx = sf.SnorkelSDKContext.from_endpoint_url(...)
sf.SnorkelSDKContext.set_global(ctx)
```

Parameters

Name	Type	Default	Info
ctx	`Optional[SnorkelSDKContext]`	`None`	A context to set as global. If no context is provided, will attempt to create one with default parameters.

Return type

`None`

property **storage_client**: *StorageClient*

property **workspace_name**: *str*

SnorkelSDKContext objects are only scoped to work in a particular workspace.

Returns

The name of the workspace belonging to this context object.

Return type

`str`

snorkelai.sdk.client.external_models

External model endpoints related functions. External models endpoints are used to configure specific models from foundation model providers.

Functions

`delete_external_model_endpoint`(model_name)	Removes an external model endpoint from DB (Only for superadmin users).
`get_external_model_endpoints`([model_name, ...])	Gets the external model endpoints from DB (Only for superadmin users).
`set_external_model_endpoint`(model_name, ...)	Adds an external model endpoint to DB (Only for superadmin users).

snorkelai.sdk.client.external_models.delete_external_model_endpoint

snorkelai.sdk.client.external_models.delete_external_model_endpoint(*model_name*)

Removes an external model endpoint from DB (Only for superadmin users).

Parameters

Name	Type	Default	Info
model_name	`str`		Name of the model.

Return type

`None`

snorkelai.sdk.client.external_models.get_external_model_endpoints

snorkelai.sdk.client.external_models.get_external_model_endpoints(*model_name=None, detail=False*)

Gets the external model endpoints from DB (Only for superadmin users).

Parameters

Name	Type	Default	Info
model_name	`Optional[str]`	`None`	Name of the model to retrieve the endpoint or configuration for. Defaults to None to return specified information for all models.
detail	`bool`	`False`	Whether to return the full configuration for each model. Defaults to False to return only the endpoint URL.

Returns

Mapping of model name to model configuration

Return type

`Dict[str, Any]`

snorkelai.sdk.client.external_models.set_external_model_endpoint

snorkelai.sdk.client.external_models.set_external_model_endpoint(*model_name, endpoint, model_provider, fm_type, **config_kwargs*)

Adds an external model endpoint to DB (Only for superadmin users). NOTE: this will impact all users who elect to use *model_name*

Parameters

Name	Type	Default	Info
model_name	`str`		Name of the model.
endpoint	`str`		Endpoint of the model.
model_provider	`str`		Name of the model provider (one of: azure_ml, azure_openai, bedrock, custom_inference_service, huggingface, openai, sagemaker, vertexai_lm).
fm_type	`str`		The model type of the foundation mode (one of: text2text, qa, docvqa).
config_kwargs	`Any`		Any additional config options to set for the model.

Return type

`None`

snorkelai.sdk.client.file_storage_configs

File storage config related functions.

Functions

`create_file_storage_config`(name, base_path)	Create a file storage config.
`delete_file_storage_config`(...)	Deletes the file storage config
`get_file_storage_config`(file_storage_config_name)	Get a specific file storage config based on the name
`get_file_storage_configs`()	Get a list of all file storage configs
`set_default_file_storage_config`(...)	Set the file storage config as the default

snorkelai.sdk.client.file_storage_configs.create_file_storage_config

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.file_storage_configs.create_file_storage_config(*name*, *base_path*)

Create a file storage config.

Parameters

Name	Type	Default	Info
name	`str`		The name of the new file storage config.
base_path	`str`		The base path to store files in. Ex: If the desired storage path is s3://snorkel-data/folder, the base path is snorkel-data/folder.

Returns

The created file storage config object

Return type

`Dict[str, Any]`

Examples

```
>>> sf.create_file_storage_config("remote", "example-bucket/inner-
folder")
{
    'file_storage_config_uid': <config_uid>,
    'name': 'remote',
    'is_default': <bool>,
    'base_path': 'example-bucket/inner-folder'
}
```

snorkelai.sdk.client.file_storage_configs.delete_file_storage_config

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.file_storage_configs.delete_file_storage_config(*file_storage_config_name*)

Deletes the file storage config

Parameters

Name	Type	Default	Info
file_storage_config_name	`str`		The name of the storage config to delete.

Return type

`None`

snorkelai.sdk.client.file_storage_configs.get_file_storage_config

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.file_storage_configs.get_file_storage_config(*file_storage_config_name*)

Get a specific file storage config based on the name

Parameters

Name	Type	Default	Info
file_storage_config_name	`str`		The name of the file storage config.

Returns

The requested file storage config object

Return type

`Dict[str, Any]`

Examples

```
>>> sf.get_file_storage_config(file_storage_config_name="minio")
{
    'file_storage_config_uid': <config_uid>,
    'name': <storage_config_name>,
    'is_default': <bool>,
    'base_path': <base_path_string>,
}
```

snorkelai.sdk.client.file_storage_configs.get_file_storage_configs

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.file_storage_configs.get_file_storage_configs()

Get a list of all file storage configs

Returns

A list of all file storage config objects

Return type

`List[Dict[str, Any]]`

Examples

```
>>> sf.get_file_storage_configs()
[
    {
        'file_storage_config_uid': <config_uid>,
        'name': <storage_config_name>,
        'is_default': <bool>,
        'base_path': <base_path_string>,
    }
    ...
]
```

snorkelai.sdk.client.file_storage_configs.set_default_file_storage_config

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.file_storage_configs.set_default_file_storage_config(*file_storage_config_name*)

Set the file storage config as the default

Parameters

Name	Type	Default	Info
file_storage_config_name	`str`		The name of the default storage config.

Return type

`None`

snorkelai.sdk.client.files

File storage related functions to upload and download files and directories.

Functions

`download_dir`(remote_path, local_path)	Downloads remote directory from Snorkel Files Store to local directory.
`download_file`(remote_path, local_path)	Downloads remote file from Snorkel Files Store to local file.
`list_dir`(remote_path)	Lists files in remote directory from Snorkel Files Store.
`upload_dir`(local_path, remote_path)	Uploads a local directory to the Snorkel Files Store.
`upload_file`(local_path, remote_path)	Uploads a file to the Snorkel Files Store.

snorkelai.sdk.client.files.download_dir

snorkelai.sdk.client.files.download_dir(*remote_path, local_path*)

Downloads remote directory from Snorkel Files Store to local directory. Files and subdirectories inside the remote directory will be placed directly in the local directory. Both absolute paths (ex. minio://workspace-1/data_dir) and relative paths (ex. data_dir, workspace-1/data_dir) are supported.

⚠ WARNING

If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/data_dir -> data_dir.

Example

```
remote_path = "minio://workspace-1/data_dir" # equivalent to "data_dir"
local_path = "/home/user/download_dir"

# Files and sub-directories under `remote_path` will be downloaded to
/home/user/download_dir
sf.download_dir(remote_path, local_path)

# To preserve the directory name, you can specify the local path like
this:
sf.download_dir(remote_path, "/home/user/download_dir/data_dir")

# These calls will raise a ValueError
sf.download_dir("minio://workspace-{not-current-workspace-id}/data_dir",
local_path)
sf.download_dir("workspace-{not-current-workspace-id}/data_dir",
local_path)
```

Parameters

Name	Type	Default	Info
remote_path	`str`		Path of remote directory to be downloaded.
local_path	`str`		Local directory to download file to.

Return type

`None`

snorkelai.sdk.client.files.download_file

snorkelai.sdk.client.files.download_file(*remote_path, local_path*)

Downloads remote file from Snorkel Files Store to local file. Both absolute paths (ex. minio://workspace-1/data/file.txt) and relative paths (ex. data/file.txt, workspace-1/data/file.txt) are supported.

⚠ WARNING

If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/file.txt -> file.txt.

Example

```
remote_path = "minio://workspace-1/data/file.txt"
local_path = "/home/user/file.txt"

# File will be downloaded to /home/user/file.txt
sf.download_file(remote_path, local_path)

# These calls will raise a ValueError
sf.download_file("minio://workspace-{not-current-workspace-
id}/data/file.txt", local_path)
sf.download_file("workspace-{not-current-workspace-id}/data/file.txt",
local_path)
```

Parameters

Name	Type	Default	Info
remote_path	`str`		Path of file to be downloaded.
local_path	`str`		Local path to download file to.

Return type

`None`

snorkelai.sdk.client.files.list_dir

snorkelai.sdk.client.files.list_dir(*remote_path*)

Lists files in remote directory from Snorkel Files Store. Both absolute paths (ex. minio://workspace-1/data_dir) and relative paths (ex. data_dir, workspace-1/data_dir) are supported.

> ⚠ **WARNING**
>
> If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/data_dir -> data_dir.

Example

```
remote_path = "minio://workspace-1/data_dir" # equivalent to "data_dir"

# Files in the remote directory will be listed
sf.list_dir(remote_path)

# These calls will raise a ValueError
sf.list_dir("minio://{not-current-workspace-id}/data_dir")
sf.list_dir("{not-current-workspace-id}/data_dir")
```

Parameters

Name	Type	Default	Info
remote_path	`str`		Path to directory to be listed.

Returns

List of files in specified remote directory

Return type

`List[Any]`

snorkelai.sdk.client.files.upload_dir

snorkelai.sdk.client.files.upload_dir(*local_path*, *remote_path*)

Uploads a local directory to the Snorkel Files Store. Both absolute paths (ex. minio://workspace-1/data_dir) and relative paths (ex. data_dir, workspace-1/data_dir) are supported.

> ⚠ **WARNING**
>
> If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/data_dir -> data_dir.

Example

```
local_path = "/home/user/data_dir"
remote_path = "data_dir"

# Directory will be uploaded to minio://workspace-{current-workspace-
id}/data_dir
sf.upload_dir(local_path, remote_path)

# These calls will raise a ValueError
sf.upload_dir(local_path, "minio://workspace-{not-current-workspace-
id}/data_dir")
sf.upload_dir(local_path, "workspace-{not-current-workspace-
id}/data_dir")
```

Parameters

Name	Type	Default	Info
local_path	`str`		Directory containing files to be uploaded.
remote_path	`str`		Remote directory on Snorkel Files Store to upload files to.

Returns

Tuple containing:

- List of uploaded file paths
- Uploaded directory path

Return type

`Tuple[List[str], str]`

snorkelai.sdk.client.files.upload_file

snorkelai.sdk.client.files.upload_file(*local_path, remote_path*)

Uploads a file to the Snorkel Files Store. Both absolute paths (ex. minio://workspace-1/file.txt) and relative paths (ex. file.txt, workspace-1/file.txt) are supported.

⚠ WARNING

If you're including the workspace prefix in the remote path, the workspace prefix (workspace-{#}) must match the current workspace. Relative paths will be resolved by removing the workspace prefix: workspace-1/file.txt -> file.txt.

Example

```
local_path = "/home/user/file.txt"
remote_path = "file.txt"

# File will be uploaded to minio://workspace-{current-workspace-id}/file.txt
sf.upload_file(local_path, remote_path)

# These calls will raise a ValueError
sf.upload_file(local_path, "minio://workspace-{not-current-workspace-id}/file.txt")
sf.upload_file(local_path, "workspace-{not-current-workspace-id}/file.txt")
```

Parameters

Name	Type	Default	Info
local_path	`str`		Path to file to be uploaded.
remote_path	`str`		File path in Snorkel Files Store to upload file to.

Returns

The uploaded file path

Return type

`str`

snorkelai.sdk.client.fm_suite

Foundation model suite related functions. Provides functions for estimating costs, viewing available methods, running jobs, and monitoring progress.

Functions

`prompt_fm`(prompt, model_name[, model_type, ...])	Send one or more prompts to a foundation model
`prompt_fm_over_dataset`(prompt_template, ...)	Run a prompt over a dataset.

snorkelai.sdk.client.fm_suite.prompt_fm

snorkelai.sdk.client.fm_suite.prompt_fm(*prompt, model_name, model_type=None, question=None, runs_per_prompt=1, sync=True, cache_name='default', **fm_hyperparameters*)

Send one or more prompts to a foundation model

Parameters

Name	Type	Default	Info
prompt	`Union[str, List[str]]`		The prompt(s) to send to the foundation model.
model_name	`str`		The name of the foundation model to use.
model_type	`Optional[LLMType]`	`None`	The way we should use the foundation model, must be one of the LLMType values.
question	`Optional[str]`	`None`	When provided, this get's passed to the model for each prompt which is useful for information retrieval tasks. The prompt argument essentially then becomes the context(s) which contains the answer to the question.

runs_per_prompt	`int`	`1`	The number of times to run a prompt, note each response can be different. All will be cached.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
cache_name	`str`	`'default'`	The cache name is used in the hash construction. To run a prompt and get a different result, you should change the cache name to something that hasn't been used before. For example: >> sf.prompt_fm("What is the meaning of life?", "openai/gpt-4o") The meaning of life is to work… >> sf.prompt_fm("What is the meaning of life?", "openai/gpt-4o") << hit's the cache The meaning of life is to work… >> sf.prompt_fm("What is the meaning of life?", "openai/gpt-4o", cache_name="run_2")

			<< hit’s a different part of the cache The meaning of life is to have fun!.
fm_hyperparameters	`Any`		Additional keyword arguments to pass to the foundation model such as temperature, max_tokens, etc.

Return type

`Union`[`DataFrame`, `str`]

Returns

- *df* – Dataframe containing the predictions for the data points. There are two columns, the input prompt and the output of the foundation model.
- *job_id* – The job id of the prompt inference job which can be used to monitor progress with sf.poll_job_status(job_id).

Examples

```
>>> sf.prompt_fm(prompt="What is the meaning of life?",
model_name="openai/gpt-4")
   | text                           | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | What is the meaning of life?   | Life is all about having fun!
| 0.789
```

```
>>> sf.prompt_fm(prompt=["What is the meaning of life?", "What is the
meaning of death?"], model_name="openai/gpt-4")
   | text                           | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | What is the meaning of life?   | Life is all about having fun!
| 0.789
1  | What is the meaning of death?  | Death is about not having fun!
| 0.981
```

```
>>> sf.prompt_fm(question="What is surname", prompt="Joe Bloggs is a
person", model_name="deepset/roberta-base-squad2")
   | text                           | answer         | start  | end   |
score
------------------------------------------------------------------------
-------------------------------------
0  | Joe Bloggs is a person         | Bloggs         | 4      | 11    |
0.985
```

snorkelai.sdk.client.fm_suite.prompt_fm_over_dataset

snorkelai.sdk.client.fm_suite.prompt_fm_over_dataset(*prompt_template, dataset, x_uids, model_name, model_type=None, runs_per_prompt=1, sync=True, cache_name='default', system_prompt=None, **fm_hyperparameters*)

Run a prompt over a dataset. Any field in the dataset can be referenced in the prompt by using curly braces, {field_name}.

Parameters

Name	Type	Default	Info
prompt_template	`str`		The prompt template used to format input rows sent to the foundation model.
dataset	`Union[str, int]`		The name or UID of the dataset containing the data we want to prompt over.
x_uids	`List[str]`		The x_uids of the rows within the dataset to prompt over.
model_name	`str`		The name of the foundation model to use.
model_type	`Optional[LLMType]`	`None`	The way we should use the foundation model, must be one of the LLMType values.

runs_per_prompt	`int`	`1`	The number of times to run inference over an xuid, note each response can be different. All will be cached.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
cache_name	`str`	`'default'`	The cache name is used in the hash construction. To run a prompt and get a different result, you should change the cache name to something that hasn't been used before. For example: >> sf.prompt_fm("What is the meaning of life?", "openai/gpt-4o") The meaning of life is to work... >> sf.prompt_fm("What is the meaning of life?", "openai/gpt-4o") << hit's the cache The meaning of life is to work... >> sf.prompt_fm("What is the meaning of life?", "openai/gpt-4o",

			cache_name=”run_2”) << hit’s a different part of the cache The meaning of life is to have fun!.
system_prompt	`Optional[str]`	`None`	The system prompt to prepend to the prompt.
fm_hyperparameters	`Any`		Additional keyword arguments to pass to the foundation model such as temperature, max_tokens, etc.

Return type

`Union[DataFrame, str]`

Returns

- *df* – Dataframe containing the predictions for the data points. There are two columns, the input prompt and the output of the foundation model.
- *job_id* – The job id of the prompt inference job which can be used to monitor progress with sf.poll_job_status(job_id).

Examples

```
>>> sf.prompt_fm_over_dataset(prompt_template="{email_subject}. What is
this email about?", dataset=1, x_uids=["0", "1"],
model_name="openai/gpt-4")
   | email_subject                          | generated_text
| perplexity
------------------------------------------------------------------------
-----------------------------------------------------------
0  | Fill in survey for $50 amazon voucher  | The email is asking you to
fill in a survey for an amazon voucher       | 0.891
1  | Hey it's Bob, free on Sat?             | The email is from your
friend Bob as if you're free on Saturday         | 0.787
```

snorkelai.sdk.client.gts

Ground truth related functions. Ground truth is the set of labels that are considered to be the "true" labels for a dataset. Ground truth is used to evaluate the performance of a model, calculate labeling function performance, and power analysis and auto-suggest tools. Ground truth labels are associated with a single model node in an application.

Functions

`add_ground_truth`(node[, x_uids, labels, df, ...])	This function registers ground truth to the Snorkel Flow platform.
`align_external_ground_truth`(node_uid, ...[, ...])	Note: This SDK is only necessary when you use a non-dataframe format with sf.add_ground_truth for sequence tagging applications.
`create_ground_truth_version`(node, name[, ...])	Create a new ground truth version for a node.
`delete_ground_truth_version`(node, gt_version_uid)	Delete a specified ground truth version for a node.
`get_document_ground_truth`(node[, context_uids])	Get document-level ground truths as Pandas DataFrame.
`get_ground_truth`(node[, split, is_context, ...])	Get ground truth from the Snorkel Flow platform.
`get_span_level_ground_truth_conflicts`(node)	Retrieve a list of span-level ground truth conflicts.
`list_ground_truth_versions`(node)	Fetch a list of all ground truth versions for a node.
`load_ground_truth_version`(node, gt_version_uid)	Load a specified ground truth version for a node.

snorkelai.sdk.client.gts.add_ground_truth

snorkelai.sdk.client.gts.add_ground_truth(*node, x_uids=None, labels=None, *, df=None, metadata=None, user_format=False, skip_missing=False, auto_generate_negative_labels=False*)

This function registers ground truth to the Snorkel Flow platform. Depending on the use case, the input format for add_ground_truth varies. Please refer to the specific section for your application type. Note that ground truth will be ingested for the particular `node` specified.

1. Single-label Applications

   Provide a list of `labels` corresponding to each data point specified by the `x_uids` argument. The labels are strings if `user_format=True`, and integers if `user_format=False`.

   ```
   >>> x_uids = ["doc::0", "doc::1", "doc::2"]
   >>> labels = ["loan", "services", "employment"]
   >>> sf.add_ground_truth(node, x_uids, labels, user_format=True)
   ```

2. Multi-label Applications

   Provide a list of `labels` corresponding to each data point specified by the `x_uids` argument. If `user_format=True`, each value in `labels` is a `dict` mapping each string label to one of `PRESENT`, `ABSENT`, or `ABSTAIN`. If `user_format=False`, each value in `labels` is a `dict` mapping each integer label to one of `PRESENT`, `ABSENT`, or `ABSTAIN`.

   ```
   >>> x_uids = ["doc::0", "doc::1", "doc::2"]
   >>> labels = [
       {"Japanese Movies": "PRESENT", "World cinema": "ABSTAIN", "Black-and-
   White": "ABSENT", "Short Film": "ABSTAIN"},
       {"Japanese Movies": "ABSTAIN", "World cinema": "PRESENT", "Black-and-
   White": "ABSTAIN", "Short Film": "PRESENT"},
       {"Japanese Movies": "ABSENT", "World cinema": "ABSENT", "Black-and-
   White": "PRESENT", "Short Film": "ABSTAIN"}
   ]
   >>> sf.add_ground_truth(node, x_uids, labels, user_format=True)
   ```

   Note that `add_ground_truth` will not accept serialized JSON. As an example for a multi-label application:

```
>>> x_uids = ["doc::0"]
>>> labels = ['{"Japanese Movies": "PRESENT", "World cinema": "ABSTAIN",
"Black-and-White": "ABSENT", "Short Film": "ABSTAIN"}']
>>> sf.add_ground_truth(node, x_uids, labels, user_format=True)
# This will fail since the labels are JSON serialized as strings
>>> deserialized_labels = [json.loads(label) for label in labels]
>>> sf.add_ground_truth(node, x_uids, deserialized_labels, user_format=True)
```

3. Sequence Tagging Applications

   Provide a dataframe containing the following columns: 1. context_uid: The unique identifier for each data point. 2. char_start: The starting position of the labeled span. 3. char_end: The ending position of the labeled span. 4. label: The label for the span.

   Starting with version 0.95, when providing a DataFrame, the labels will be aligned automatically (no need to run `align_external_ground_truth`).

   > context_uid | char_start | char_end | label doc::0 | 0 | 10 | COMPANY doc::1 | 0 | 5 | COMPANY doc::1 | 5 | 15 | OTHER

   ```
   >>> df = pd.DataFrame({
       "context_uid": ["doc::0", "doc::1", "doc::1"],
       "char_start": [0, 0, 5],
       "char_end": [10, 5, 15],
       "label": ["COMPANY", "COMPANY", "OTHER"]
   })
   >>> sf.add_ground_truth(node, df=df)
   ```

4. Sequence Tagging Applications (legacy)

   Provide a list of `labels` corresponding to each data point specified by the `x_uids` argument. Each value in `labels` is a list of spans corresponding to each data point. The spans are formatted as `(start, end, label)` where `start` and `end` are the starting and ending indices of the span, and `label` is the label of the span. If `user_format=True`, the labels are strings. If `user_format=False`, the labels are integers.

   ```
   >>> x_uids = ["doc::0", "doc::1", "doc::2"]
   >>> labels = [
       [(0, 10, "COMPANY"), (15, 20, "OTHER")],
       [(0, 5, "COMPANY")],
       [(5, 15, "COMPANY")]
   ]
   >>> sf.add_ground_truth(node, x_uids, labels, user_format=True)
   ```

If using the traditional input format (i.e., `x_uids` + `labels`), you must still run `align_external_ground_truth` to align the ground truth spans after preprocessing.

To learn more about the Label format for your use case you can see Format for ground truth interaction in the SDK.

Parameters

Name	Type	Default	
node	`int`		UID of the node for.
x_uids	`Union[List[str], ndarray, None]`	`None`	UIDs of data poi numpy array of
labels	`Union[List[Any], ndarray, None]`	`None`	Label values. Lis labels. Must be t x_uids. If user_fo labels have not l user_format is F instead of string
df	`Optional[DataFrame]`	`None`	Specific for sequ pandas datafran following colum start, end, label provide one of (`df`
metadata	`Optional[Dict[str, Any]]`	`None`	Metadata to reg (information abo was generated).
user_format	`bool`	`False`	True if the grour provided in user To see a mappir integer represer (for `user_form` `sf.get_node_`

skip_missing	`bool`	`False`	Specific for legac cases. True if sp datasource need otherwise.
auto_generate_negative_labels	`bool`	`False`	Specific for legac cases. If True, au negative ground

Return type

`None`

snorkelai.sdk.client.gts.align_external_ground_truth

snorkelai.sdk.client.gts.align_external_ground_truth(*node_uid, x_uids, labels, user_format=False, scheduler=None*)

Note: This SDK is only necessary when you use a non-dataframe format with sf.add_ground_truth for sequence tagging applications. Starting with version 0.95, if you use a dataframe with sf.add_ground_truth, the labels will automatically be aligned

(Sequence Tagging Only) This function changes external ground truth spans to compensate for the offsets caused by a text preprocessor.

Text Preprocessors may sometimes remove characters, resulting in misalignments between externally collected ground truth spans and the preprocessed text. For example, the default `AsciiCharFilter` preprocessor removes non-ascii characters which will cause some spans to shift leftwards, but external annotations will still have the original spans. Thus, it is necessary to use this function for applications with non-ascii characters.

Examples

An example for a sequence tagging application with `AsciiCharFilter` as the preprocessor

```
x_uids = ["doc::0", "doc::1"] # all x_uids with labels
labels = [
    [[0, 20, "COMPANY"]], # labels for doc::0
    [[10, 15, "COMPANY"], [20, 25, "COMPANY"]], # labels for doc::1
    ...
]

aligned_labels = sf.align_external_ground_truth(node_uid, x_uids,
labels, user_format=True)
```

Then, use `sf.add_ground_truth` to add `aligned_labels`

```
sf.add_ground_truth(node_uid, x_uids, aligned_labels, user_format=True)
```

Parameters

Name	Type	Default	Info

node_uid	`int`		The UID of the model node.
x_uids	`List[str]`		UIDs of data points. Can be a list or a 1D numpy array of strings.
labels	`List[Any]`		Label values. List or numpy array of labels. Must be the same length as x_uids. If user_format is True, check that labels have not been JSON serialized.
user_format	`bool`	`False`	True if labels are provided in user format, False otherwise.
scheduler	`Optional[str]`	`None`	Dask scheduler (threads, client, or group) to use.

Returns

List of aligned labels corresponding to x_uids.

Return type

`List[Any]`

snorkelai.sdk.client.gts.create_ground_truth_version

snorkelai.sdk.client.gts.create_ground_truth_version(*node, name, description=<snorkelai.sdk.client_v3.tdm.types.Unset object>*)

Create a new ground truth version for a node. Creating a ground truth version will yield an integer UID that identifies the ground truth version. The `snorkelai.sdk.client.load_ground_truth_version` function can then be used to restore a previously created ground truth version. Note that ground truth versions are only available in the node the ground truth version was created for.

Examples

```
>>> sf.create_ground_truth_version(node=1, name="my_gt_version",
description="my description")
{'gt_version_uid': 1}
```

Parameters

Name	Type	Default
node	`int`	
name	`str`	
description	`Union[Unset, str]`	`<snorkelai.sdk.client_v3.tdm.types.Unset object at 0x7b14f3aef5c0>`

Returns

gt_version_uid: The UID of the new ground truth version

Return type

`Dict[str, int]`

snorkelai.sdk.client.gts.delete_ground_truth_version

snorkelai.sdk.client.gts.delete_ground_truth_version(*node, gt_version_uid*)

Delete a specified ground truth version for a node. This is a destructive operation, and it will not be possible to load this ground truth version once this operation is complete.

Examples

```
>>> sf.list_ground_truth_versions(node=1)
[{'gt_version_uid': 1, ... }]
>>> sf.delete_ground_truth_version(node=1, gt_version_uid=1)
>>> sf.list_ground_truth_versions(node=1)
[]
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node to delete the GT version of.
gt_version_uid	`int`		The UID of the ground truth version to delete.

Return type

`None`

snorkelai.sdk.client.gts.get_document_ground_truth

snorkelai.sdk.client.gts.get_document_ground_truth(*node*, *context_uids=None*)

Get document-level ground truths as Pandas DataFrame.

This function gets the ground truth for an entire input document. It can only be used for information extraction tasks. For text extraction tasks, this gets a list of extractions for each document. For entity classification tasks, this gets a dictionary mapping entities to classes for each document. The ground truth is always represented as integers rather than strings. A mapping from integer to string labels can be retrieved using `sf.get_node_label_map`. For information about how to interpret ground truth in information extraction tasks, see Format for ground truth interaction in the SDK.

Examples

```
>>> sf.get_document_ground_truth(123, context_uids=[456, 789])
<pd.DataFrame>
__DATAPOINT_UID      |   ground_truth
doc::0               |   [[0, 10, 0]]
doc::1               |   [[0, 5, 1], [5, 15, 0]]
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node whose document ground truth to get.
context_uids	`Optional[List[int]]`	`None`	Optional list of context_uids of documents whose ground truth to get. If None, get all document ground truth.

Return type

`DataFrame`

Returns

- *Pandas DataFrame containing document ground truth with*
- *document context_uid as index*

snorkelai.sdk.client.gts.get_ground_truth

snorkelai.sdk.client.gts.get_ground_truth(*node, split=None, is_context=False, user_format=False*)

Get ground truth from the Snorkel Flow platform. The format of the ground truth depends on the task type and whether `user_format` is True or False. For more information, see Format for ground truth interaction in the SDK.

Examples

```
>>> sf.get_ground_truth(node=1, split='train', user_format=True)
<pd.Series>
doc::0      SPAM
doc::1      HAM
doc::2      HAM
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node whose ground-truth we're fetching.
split	`Optional[str]`	`None`	Name of a split whose ground-truth to fetch (if None, return all splits).
is_context	`bool`	`False`	If False, get datapoint level ground truth. Otherwise, get context level ground truth. Only relevant for information extraction tasks.
user_format	`bool`	`False`	True if the ground truth labels are returned in user format, False otherwise.

Returns

A pandas Series containing ground truth labels.

Return type

`pd.Series`

snorkelai.sdk.client.gts.get_span_level_ground_truth_conflicts

snorkelai.sdk.client.gts.get_span_level_ground_truth_conflicts(*node, split=None*)

Retrieve a list of span-level ground truth conflicts. The conflicts are sorted by entropy. For a span_text where all ground truth labels are the same, the entropy is 0. For a span_text with multiple different ground truth labels, the entropy > 0. The larger entropy indicates more conflicts.

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node whose ground-truth we're fetching.
split	`Optional[str]`	`None`	Name of a split whose ground-truth to fetch (if None, return all splits).

Returns

List of dict of span-level conflicts in a DataFrame, sorted by entropy, i.e., the span with most conflicting ground truth labels comes first

Return type

`pd.DataFrame`

snorkelai.sdk.client.gts.list_ground_truth_versions

snorkelai.sdk.client.gts.list_ground_truth_versions(*node*)

Fetch a list of all ground truth versions for a node. The return list will reveal the corresponding ground truth version UIDs for all ground truth versions for this node, which can then be used in `snorkelai.sdk.client.load_ground_truth_version` and `snorkelai.sdk.client.delete_ground_truth_version.`

Examples

```
>>> sf.list_ground_truth_versions(node=1)
[
    {
        'gt_version_uid': 1,
        'node_uid': 1,
        'name': 'my-gt-version',
        'description': 'my description',
        'hidden': False,
        'created_by': 'notebook',
        'created_at': '2023-07-10T20:33:09.323068'
    }
]
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node to fetch all GT versions for.

Returns

Returns a list of ground truth versions for this node, displaying the information relevant to each ground truth version object.

Return type

`List[Dict[str, Any]]`

snorkelai.sdk.client.gts.load_ground_truth_version

snorkelai.sdk.client.gts.load_ground_truth_version(*node*, *gt_version_uid*)

Load a specified ground truth version for a node. This will change the ground truth that you see in Studio. Make sure to save your previous ground truth version with `snorkelai.sdk.client.create_ground_truth_version` before loading a new one to avoid losing any ground truth labels. You can call `sf.poll_job_status` on the `job_uid` returned by this function to track the progress of this operation.

The ground truth labels you load will become available in Developer Studio once this function has finished executing. Keep in mind that you may need to resample the dev split of your selected node to see the new ground truth labels.

Examples

```
>>> sf.load_ground_truth_version(node=1, gt_version_uid=1)
{'job_uid': 'rq-ZZyIRDE7_engine-D2nP_load-ground-truth-version'}
>>> sf.poll_job_status('rq-ZZyIRDE7_engine-D2nP_load-ground-truth-
version')
+0.31s Finished loading ground truth version
# ... ground truth labels are now available in Developer Studio after
resampling the Dev split
```

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node to load a GT version for.
gt_version_uid	`int`		The UID of the ground truth version to load.

Return type

`AsyncJobResponse`

snorkelai.sdk.client.secrets

Secret store related functions.

Functions

`delete_secret`(key[, secret_store, ...])	Deletes a secret from the secret store (Only for superadmin users).
`get_model_provider_status`(model_provider)	Gets the status of a model provider.
`list_integrations`([secret_store, ...])	Gets configuration status for each foundation model provider (Only for superadmin users).
`list_secrets`([secret_store, workspace_uid, ...])	Gets all secret keys in a workspace (Only for superadmin users).
`set_secret`(key, value[, secret_store, kwargs])	Adds secret to the secret store (Only for superadmin users).

snorkelai.sdk.client.secrets.delete_secret

snorkelai.sdk.client.secrets.delete_secret(*key, secret_store='local_store', workspace_uid=1, kwargs=None*)

Deletes a secret from the secret store (Only for superadmin users).

Parameters

Name	Type	Default	Info
key	`str`		Key to reference the secret in the store.
secret_store	`str`	`'local_store'`	The secret store to delete the secret (only *local_store* supported now).
workspace_uid	`int`	`1`	The workspace uid for the secret.
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Return type

`None`

snorkelai.sdk.client.secrets.get_model_provider_status

snorkelai.sdk.client.secrets.get_model_provider_status(*model_provider*)

Gets the status of a model provider.

Parameters

Name	Type	Default	Info
model_provider	`ExternalLLMProvider`		The model provider to retrieve the status for.

Returns

The status of the model provider

Return type

`Dict[str, Any]`

snorkelai.sdk.client.secrets.list_integrations

snorkelai.sdk.client.secrets.list_integrations(*secret_store='local_store', workspace_uid=1, kwargs=None*)

Gets configuration status for each foundation model provider (Only for superadmin users).

Parameters

Name	Type	Default	Info
secret_store	`str`	`'local_store'`	The secret store to list the secret (only *local_store* supported now).
workspace_uid	`int`	`1`	The workspace uid for the secret.
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Return type

`None`

snorkelai.sdk.client.secrets.list_secrets

snorkelai.sdk.client.secrets.list_secrets(*secret_store='local_store', workspace_uid=1, kwargs=None*)

Gets all secret keys in a workspace (Only for superadmin users).

Parameters

Name	Type	Default	Info
secret_store	`str`	`'local_store'`	The secret store to list the secret (only *local_store* supported now).
workspace_uid	`int`	`1`	The workspace uid for the secret.
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Returns

All secret keys associated with a workspace

Return type

`List[str]`

snorkelai.sdk.client.secrets.set_secret

snorkelai.sdk.client.secrets.set_secret(*key, value, secret_store='local_store', kwargs=None*)

Adds secret to the secret store (Only for superadmin users).

Parameters

Name	Type	Default	Info
key	`str`		Key to reference the secret in the store.
value	`Union[str, Dict[str, str]]`		The secret being added.
secret_store	`str`	`'local_store'`	The secret store to add the secret (only *local_store* supported now).
kwargs	`Optional[Dict[str, Any]]`	`None`	Other connection kwargs for accesing the secret store.

Return type

`None`

snorkelai.sdk.client.synthetic

Synthetic data related functions for generating synthetic data.

Functions

`augment_data`(data, model_name[, ...])	Augment each row of the data by the number of times specified and return a dataframe with the synthetic data as an additional column.
`augment_dataset`(dataset, x_uids, model_name)	Augment each row of the dataset by the number of times specified and return a dataframe containing only the synthetic data.

snorkelai.sdk.client.synthetic.augment_data

snorkelai.sdk.client.synthetic.augment_data(*data, model_name, runs_per_prompt=1, prompt='Rewrite the following text whilst retaining the core meaning.', sync=True, **fm_hyperparameters*)

Augment each row of the data by the number of times specified and return a dataframe with the synthetic data as an additional column.

Parameters

Name	Type	Default	Info
data	`Union[List[str], str]`		The data to augment.
model_name	`str`		The name of the foundation model to use.
runs_per_prompt	`int`	`1`	The number of times to augment each row.
prompt	`str`	`'Rewrite the following text whilst retaining the core meaning.'`	The prompt prefix to send to the foundation model together with each row.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
fm_hyperparameters	`Any`		Additional keyword arguments to pass

			to the foundation model such as temperature, max_tokens, etc.

Return type

`Union`[`DataFrame`, `str`]

Returns

- *df* – Dataframe containing the augmentations for the data points.
- *job_id* – The job id of the augment data job which can be used to monitor progress with sf.poll_job_status(job_id).

Examples

```
>>> sf.augment_data(["hello, how can I help you?", "sorry that is not
possible"], "openai/gpt-4")
   | text                             | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | hello, how can I help you?     | welcome, ask me a question to get
started   | 0.0113636364
1  | sorry that is not possible     | unfortunately you cannot do that
| 0.8901232123
```

```
>>> sf.augment_data(["hello, how can I help you?", "sorry that is not
possible"], "openai/gpt-4", runs_per_prompt=2)
   | text                            | generated_text
| perplexity
------------------------------------------------------------------------
--------------------------------------
0  | hello, how can I help you?     | welcome, ask me a question to get
started   | 0.0113636364
1  | sorry that is not possible     | unfortunately you cannot do that
| 0.8901232123
0  | hello, how can I help you?     | Let me know how to get started.
| 0.2313232442
1  | sorry that is not possible     | bad luck, you cannot do that.
| 0.8313232442
```

snorkelai.sdk.client.synthetic.augment_dataset

snorkelai.sdk.client.synthetic.augment_dataset(*dataset, x_uids, model_name, runs_per_prompt=1, prompt='Your task is to rewrite the a set of text fields whilst retaining the core meaning. You should keep the same language and ensure each re-written field is of a similar length to the original.', fields=None, sync=True, **fm_hyperparameters*)

Augment each row of the dataset by the number of times specified and return a dataframe containing only the synthetic data. By default, all fields are augmented and the foundation model performs the augmentation of each row (all fields) in one inference step.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The name or UID of the dataset to generate a synthetic augmentation of.
x_uids	`List[str]`		The x_uids within the dataset to augment.
model_name	`str`		The name of the foundation model to use.
runs_per_prompt	`int`	`1`	The number of times to augment each row.
prompt	`str`	`'Your task is to`	The prompt passed to the

		`rewrite the a set of text fields whilst retaining the core meaning. You should keep the same language and ensure each re-written field is of a similar length to the original.'`	foundation model for each row. Note that by default, the prompt is appended with the fields to make the following: "Rewrite the following text fields whilst retaining the core meaning. You should keep the same language and ensure each re-written field is of a similar length to the original. Return your answer in a json format with the same keys as the fields: [field_1, field_2, ...] Here is the data you have to rewrite...". To override this default behavior, simply pass at least one field wrapped in parentheses, e.g. {field_1}, within the

			prompt and no additional text will be append to the prompt.
fields	`Optional[List[str]]`	`None`	The fields to augment. If not provided, all fields will be augmented.
sync	`bool`	`True`	Whether to wait for the job to complete before returning the result.
fm_hyperparameters	`Any`		Additional keyword arguments to pass to the foundation model such as temperature, max_tokens, etc.

Return type

`Union`[`DataFrame`, `str`]

Returns

- *df* – Dataframe containing the augmentations for the data points.
- *job_id* – The job id of the augment data job which can be used to monitor progress with sf.poll_job_status(job_id).

Examples

```
>>> sf.augment_dataset(dataset=1, x_uids=["0", "1"],
model_name="openai/gpt-4", runs_per_prompt=2)
   | subject                                | body
| perplexity
------------------------------------------------------------------------
-----------------------------------------------------------
0  | Fill in survey for $50 amazon voucher  | The email is asking you to
fill in a survey for an amazon voucher       | 0.891
1  | Hey it's Bob, free on Sat?             | The email is from your
friend Bob asking if you're free on Saturday     | 0.787
0  | Free survey for $50                    | Want a free $50 amazon
voucher? Fill in our survey.                     | 0.911
1  | No Plans on Sat, Bob?                  | Let's meet up on Sat. Bob.
| 0.991
```

snorkelai.sdk.client.transfer

SDK functions for transferring assets between applications or nodes on a single Snorkel Flow instance.

Functions

`convert_span_gt_csv_to_span_format`(...[, ...])	Convert exported span ground truth CSV files to old span format for backwards compatibility.
`export_ground_truth`(node, filepath[, is_context])	Exports ground truth.
`export_tag_types`(node[, ...])	Export the tag types for a node, optionally to a file.
`import_ground_truth`(node, filepath[, ...])	Import ground truth labels for any task type.
`import_tag_types`(node, all_tag_types_dict[, ...])	Import `tag_types`.
`transfer_comments`(node, from_node[, merge_type])	Transfer comments from the node with ID `from_node` to the node with ID `node`.
`transfer_tags`(node, from_node[, ...])	Copy tags from `from_node` to `node`.

snorkelai.sdk.client.transfer.convert_span_gt_csv_to_span_format

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.convert_span_gt_csv_to_span_format(*gt_csv_path, converted_gt_csv_path, x_uid_col='x_uid', label_col='label'*)

Convert exported span ground truth CSV files to old span format for backwards compatibility.

Parameters

Name	Type	Default	Info
gt_csv_path	`str`		Path to ground truth CSV file for spans containing x_uid column to convert.
converted_gt_csv_path	`str`		Path to write converted CSV, containing span columns and labels.
x_uid_col	`str`	`'x_uid'`	Column name containing x_uids in the CSV file at gt_csv_path.
label_col	`str`	`'label'`	Column name containing labels in the CSV file at gt_csv_path.

Return type

`None`

snorkelai.sdk.client.transfer.export_ground_truth

WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.export_ground_truth(*node, filepath, is_context=False*)

Exports ground truth. Returns a two-column table, where the columns are the `x_uid` and `label`. Also writes this table to a CSV file at the provided location.

Examples

```
>>> sf.export_ground_truth(node_uid, gt_filepath)
# returns a pd.DataFrame
            x_uid         label
0           doc::001      "SPAM"
1           doc::002      "HAM"
```

Parameters

Name	Type	Default	Info
node	`int`		ID of the node to export GT from.
filepath	`str`		Path to the ground truth file we export to. This path must be a path in the local filesystem, i.e. cannot be in S3 or MinIO.
is_context	`bool`	`False`	If True, export a ground truth label for the whole document, if False then just for the span. Only available for information extraction tasks.

Return type

`None if there is no gt, otherwise, a pd.DataFrame with columns x_uid and label.`

snorkelai.sdk.client.transfer.export_tag_types

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.export_tag_types(*node, is_context_tag_type=None, filepath=None*)

Export the tag types for a node, optionally to a file. Tag types describe the set of possible tags that can be assigned to a document at this node. Also exports a `tag_map`, a mapping that points individual documents to the corresopnding tag type UID. You can retrieve a human-readable list of tags by replacing the values in `tag_map` with the corresponding tag name in the `tag_types` list.

Examples

```
>>> sf.export_tag_types(node_uid)
{
    'tag_types': [
        {
            'tag_type_uid': 1,
            'name': "my first tag",
            'description': "this is my first tag!",
            'is_context_tag_type': False
            }
    ],
    'tag_map': {
        'doc::1': [1, ...],
        'doc::2': [1],
    },
}
```

Parameters

Name	Type	Default	Info
node	`int`		ID of the node for which we're exporting ground truth.

is_context_tag_type	`Optional[bool]`	`None`	If True, export only context tag types. If False, export only non-context tag types. If None, export all context tag types. This option only works with non-classification tasks.
filepath	`Optional[str]`	`None`	Path to the file where the tag data is saved. Only local path is supported.

Returns

A dictionary with two keys “tag_types” and “tag_map” “tag_types” value is of type List[Dict[str, Any], which is a list of dictionary configs of tag_types.

”tag_map” value is of type Dict[str, List[int]], mapping x_uids to the associated list of tag_type_uids.

This dictionary can be used for *import_tag_types*.

Return type

`Dict[str, Any]`

snorkelai.sdk.client.transfer.import_ground_truth

WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.import_ground_truth(*node, filepath, updated_label_schema=None, file_format='CSV', label_col='label', uid_col='x_uid', run_async=False, is_context=False, auto_generate_negative_labels=False, merge_type='FROM'*)

Import ground truth labels for any task type.

The file is read as a dataframe. It is expected to have 2 columns, `x_uid` and `label`, where `x_uid` consists of `x_uids` (unique identifiers) and `label` consists of corresponding ground truth values. The default names of the required columns are `x_uid` and `label`, however, these names can be updated by passing in the *uid_col* and *label_col* parameters.

Alternatively, for span-based tasks, It is expected to have 5 columns: `label`, `context_uid`, `span_field`, `char_start`, `char_end`. The last 4 values must match those of a span for the ground truth to be updated. Different column names for the `label` can be updated by passing in the `label_col` parameters. If using this format, `uid_col` parameter must be set to None.

Examples

gt_filepath.csv: x_uid,label doc::001,SPAM doc::002,HAM

```
>>> sf.import_ground_truth(node_uid, "gt_filepath.csv")
2 # Returns the number of ground truth labels successfully imported
```

Parameters

Name	Type	Default	Info
node	`int`		ID of the node t we're uploading truth.

filepath	`str`		Path to the grou file we're uploa Both both local MinIO path are supported.
updated_label_schema	`Optional[Dict[str, str]]`	`None`	A dictionary tha remap labels fr file into labels i target node. Th for this dictiona the unique labe in the `label` c the file that is b imported. The v this dictionary a corresponding names in the destination noc example, if the contains the lab but the destina node uses the l "animal", then t dictionary shou {"dog": "animal"
file_format	`str`	`'CSV'`	Format of the g truth file, either `parquet`.
label_col	`Optional[str]`	`'label'`	Name of the co the ground trut containing the g truth value. Def "label".

uid_col	`Optional[str]`	`'x_uid'`	Name of the co the ground trut containing uid. is required for classification ta can be blank fo task types. Defa “x_uid”.
run_async	`Optional[bool]`	`False`	If true, runs the asynchronously returns a job ID be polled with `sf.poll_job_` Otherwise bloc by default.
is_context	`bool`	`False`	If True, import truth at the doc level, otherwise span level. Only applicable for information ext tasks.
auto_generate_negative_labels	`bool`	`False`	If True, fills in “ sequence taggi with the “negat “other” class. O applicable for s tagging tasks. D False.
merge_type	`str`	`'FROM'`	Specifies how t conflicts betwe imported file ar destination noc either be `TO` or

			`TO`, any conflic take the values destination noc `FROM` any confl take the values imported file.

Return type

`Union`[`int`, `str`]

Returns

- *int* – An int for how many ground truth uids were successfully updated.
- *str* – If the import job is running asynchronously, the job_uid is returned instead

snorkelai.sdk.client.transfer.import_tag_types

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.import_tag_types(*node, all_tag_types_dict, tag_types_to_import=None*)

Import `tag_types`. The format of the input dictionary `all_tag_types_dict` should match the output format of `export_tag_types`.

Examples

```
>>> my_export_dict = sf.export_tag_types(123)
>>> sf.import_tag_types(456, my_export_dict)
```

Parameters

Name	Type	Default	Info
node	`int`		ID of the node to which we’re uploading tag type.
all_tag_types_dict	`Dict[str, Any]`		A dictionary with two keys “tag_types” and “tag_map” “tag_types” value is of type List[Dict[str, Any], which is a list of dictionary configs of tags_type. ”tag_map” value is of type Dict[str, List[int]], mapping x_uids to the associated list of tag_type_uids.

			This dictionary can be obtained through *export_tag_types*.
tag_types_to_import	`Optional[List[str]]`	`None`	List of tag_types to import. If None, import all tag_types. If [], import no tag types.

Return type

`None`

snorkelai.sdk.client.transfer.transfer_comments

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.transfer_comments(*node, from_node, merge_type='FROM'*)

Transfer comments from the node with ID `from_node` to the node with ID `node`.

Examples

```
>>> sf.transfer_comments(123, 456)
# Transfer comments from 456 to 123
>>> sf.transfer_comments(123, 456, merge_type="UNION")
# Transfer comments from 456 to 123, taking the union of all comments
from both nodes
>>> sf.transfer_comments(123, 456, merge_type="FROM")
# Transfer comments from 456 to 123, taking the comments from 456
```

Parameters

Name	Type	Default	Info
node	`int`		ID of the node comments are copied to.
from_node	`int`		ID of the node comments are copied from.
merge_type	`str`	`'FROM'`	Merge type for comments, can be one of {'UNION','FROM'}. If 'UNION', comments are transferred without any special identifier. If 'FROM', comment body is prefixed with a header containing information about which node the comment was transferred from.

Return type

`None`

snorkelai.sdk.client.transfer.transfer_tags

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.transfer.transfer_tags(*node, from_node, tag_types_to_import=None, is_context_tag_type=None*)

Copy tags from `from_node` to `node`.

Examples

```
>>> sf.transfer_tags(node_uid, from_node_uid)
# Transfer all tags from from_node_uid to node_uid
>>> sf.transfer_tags(node_uid, from_node_uid, tag_types_to_import=["my
first tag"])
# Transfer only the tag "my first tag" from from_node_uid to node_uid
```

Parameters

Name	Type	Default	Info
node	`int`		ID of the node to which we're copying tags.
from_node	`int`		ID of the node from which we're copying tags.
tag_types_to_import	`Optional[List[str]]`	`None`	List of tag_types from the old node that should be imported. If None, import all tag_types. If [], import no tag_types.
is_context_tag_type	`Optional[bool]`	`None`	If True, transfer only context tag_types. If

			False, transfer only non-context tag_types. If None, transfer all context tag_types. This option is only applicable for information extraction tasks. Context tag_types are tags at the document-level, whereas other tags are at the span level.

Return type

`None`

snorkelai.sdk.client.users

User related functions.

Functions

`get_user`(user)	Get a user info by its username or user_uid.
`reset_password`(username, new_password)	Reset the password for the specified user to the specified new password.

snorkelai.sdk.client.users.get_user

snorkelai.sdk.client.users.get_user(*user*)

Get a user info by its username or user_uid.

Example

```
>>> sf.get_user("username")
{
    'username': 'username',
    'user_uid': 4,
    'default_view': 'standard',
    'role': 'standard',
    'is_active': True,
    'is_locked': False,
    'email': None,
    'timezone': None,
    'is_superadmin': False
}
```

Parameters

Name	Type	Default	Info
user	`Union[str, int]`		A valid Snorkel Flow user's username or user_uid.

Returns

The user info corresponding to the provided username/user_uid

Return type

`Dict[str, Any]`

snorkelai.sdk.client.users.reset_password

snorkelai.sdk.client.users.reset_password(*username, new_password*)

Reset the password for the specified user to the specified new password.

This functionality is only available to administrators as determined by the API key of the caller.

Parameters

Name	Type	Default	Info
username	`str`		Username of the user whose password you wish to reset.
new_password	`str`		The new password value you wish to set for this user.

Return type

`None`

snorkelai.sdk.client.utils

Utility functions.

Functions

`get_application_uid`(name)	Fetch the UID of an Application by name
`get_batch_uid`(node, batch_name)	Translate a batch_name to a batch_uid.
`get_dataset_name`(dataset_uid)	Fetch the UID of a Dataset by name
`get_dataset_uid`(dataset)	Fetch the UID of a dataset by name or UID
`get_file_storage_config_uid`(...)	Fetch the UID of a file storage config by name
`get_lf_uid`(node, lf_name)	Fetch the UID of an active LF by name.
`get_operator_uid`(operator_name)	Fetch the UID of an Operator by name.
`get_source_uid`(source_name)	Translate a source_name to a source_uid.
`get_tag_type_uid`(node, name[, ...])	Look up the tag_type_uid of a tag type by name.
`get_workspace_name`(workspace_uid)	Fetch the name of a Workspace by UID
`get_workspace_uid`(workspace_name)	Fetch the UID of a Workspace by name
`poll_job_status`(job_id[, timeout, verbose])	Poll /jobs endpoint and print statuses.
`validate_traces`(trace_csv_file_path, ...)	Validate the traces in the trace CSV file.

snorkelai.sdk.client.utils.get_application_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_application_uid(*name*)

Fetch the UID of an Application by name

Parameters

Name	Type	Default	Info
name	`str`		Name of the application.

Returns

UID of the application

Return type

`int`

snorkelai.sdk.client.utils.get_batch_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_batch_uid(*node, batch_name*)

Translate a batch_name to a batch_uid.

Parameters

Name	Type	Default	Info
node	`int`		The UID of the node.
batch_name	`str`		A valid batch name.

Returns

The batch_uid for batch_name

Return type

`int`

snorkelai.sdk.client.utils.get_dataset_name

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_dataset_name(*dataset_uid*)

Fetch the UID of a Dataset by name

Parameters

Name	Type	Default	Info
dataset_uid	`int`		UID of the dataset.

Returns

Name of the dataset

Return type

`str`

snorkelai.sdk.client.utils.get_dataset_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_dataset_uid(*dataset*)

Fetch the UID of a dataset by name or UID

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		Name or UID of the dataset.

Returns

UID of the dataset

Return type

`int`

Raises

ValueError – If a dataset doesn’t exist.

snorkelai.sdk.client.utils.get_file_storage_config_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_file_storage_config_uid(*file_storage_config_name*)

Fetch the UID of a file storage config by name

Parameters

Name	Type	Default	Info
file_storage_config_name	`str`		Name of the file storage config.

Returns

UID of the file storage config

Return type

`int`

snorkelai.sdk.client.utils.get_lf_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_lf_uid(*node*, *lf_name*)

Fetch the UID of an active LF by name.

Parameters

Name	Type	Default	Info
node	`int`		UID of the node.
lf_name	`str`		Name of the LF.

Returns

UID of the LF

Return type

`int`

snorkelai.sdk.client.utils.get_operator_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_operator_uid(*operator_name*)

Fetch the UID of an Operator by name.

Parameters

Name	Type	Default	Info
operator_name	`str`		Name of the operator whose UID is desired.

Returns

The UID of the Operator specified by operator_name

Return type

`int`

Raises

OperatorNotFound – If the Operator does not exist in the dataset

snorkelai.sdk.client.utils.get_source_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_source_uid(*source_name*)

Translate a source_name to a source_uid.

Parameters

Name	Type	Default	Info
source_name	`str`		A valid source name.

Returns

The source_uid for source_name

Return type

`int`

snorkelai.sdk.client.utils.get_tag_type_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_tag_type_uid(*node, name, is_context_tag_type=False*)

Look up the tag_type_uid of a tag type by name.

Parameters

Name	Type	Default	Info
node	`int`		UID of the node that the tag type belongs to.
name	`str`		Name of the tag type whose tag_type_uid will be returned.
is_context_tag_type	`bool`	`False`	If True, this is a context-level tag type instead of a datapoint-level tag type.

Returns

The tag_uid of the requested tag type

Return type

`int`

snorkelai.sdk.client.utils.get_workspace_name

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_workspace_name(*workspace_uid*)

Fetch the name of a Workspace by UID

Parameters

Name	Type	Default	Info
workspace_uid	`int`		UID of the workspace.

Returns

Name of the workspace

Return type

`str`

snorkelai.sdk.client.utils.get_workspace_uid

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.get_workspace_uid(*workspace_name*)

Fetch the UID of a Workspace by name

Parameters

Name	Type	Default	Info
workspace_name	`str`		Name of the workspace.

Returns

UID of the workspace

Return type

`int`

snorkelai.sdk.client.utils.poll_job_status

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.client.utils.poll_job_status(*job_id, timeout=None, verbose=True*)

Poll /jobs endpoint and print statuses.

Parameters

Name	Type	Default	Info
job_id	`str`		UID of the job.
timeout	`Optional[timedelta]`	`None`	Optional polling timeout duration, jobs that exceed the timeout will be canceled.
verbose	`bool`	`True`	Optional argument to increase verbosity of the output logs.

Returns

Final job status response if job completes

Return type

`Dict[str, Any]`

Raises

- **JobFailedException** – If job fails
- **JobCancelledException** – If job is cancelled by user

Examples

```
>>> sf.poll_job_status(job_id)
{
    'uid': <job_id>,
    'job_type': <job_type>,
    'state': <state>, # completed or failed
    'enqueued_time': <timestamp>,
    'execution_start_time': <timestamp>,
    'end_time': <timestamp>,
    'application_uid':<application_uid>,
    'dataset_uid': <dataset_uid>,
    'node_uid': <node_uid>,
    'user_uid':<user_uid>,
    'workspace_uid': <workspace_uid>,
    'percent': <percent>, # Based on state
    'message':  <message>,
    'detail': <detail>,
    'pod_name':  <pod_name>,
    'function_name':  <function_name>,
    'process_id':  <process_id>,
    'timing': <timing_dict>
}
```

snorkelai.sdk.client.utils.validate_traces

snorkelai.sdk.client.utils.validate_traces(*trace_csv_file_path, trace_column*)

Validate the traces in the trace CSV file.

Parameters

Name	Type	Default	Info
trace_csv_file_path	`str`		The path to the trace CSV file.
trace_column	`str`		The column name containing the trace JSON data.

Returns

A dictionary containing the validation results.

Return type

`Dict[str, Any]`

snorkelai.sdk.develop

(Beta) Object-oriented SDK for Snorkel Flow

Classes

`Batch`(name, uid, dataset_uid, label_schemas, ...)	The Batch object represents an annotation batch in Snorkel Flow.
`Benchmark`(*args, **kwargs)	A benchmark is the collection of characteristics that you care about for a particular GenAI application, and the measurements you use to assess the performance against those characteristics.
`BenchmarkExecution`(*args, **kwargs)	Represents a single execution run of a benchmark for a dataset.
`CodeEvaluator`(*args, **kwargs)	An evaluator that uses custom Python code to assess an AI application's responses.
`Criteria`(*args, **kwargs)	A criteria represents a specific characteristic or feature being evaluated as part of a benchmark.
`CsvExportConfig`([sep, quotechar, escapechar])	
`Dataset`(name, uid, mta_enabled)	The Dataset object represents a dataset in Snorkel Flow.
`Evaluator`(*args, **kwargs)	Base class for all evaluators.
`JsonExportConfig`()	
`LabelSchema`(name, uid, dataset_uid, ...[, ...])	The LabelSchema object represents a label schema in Snorkel Flow.
`Node`(uid, application_uid, config)	The Node object represents atomic data processing units in Snorkel Flow.
`OperatorNode`(uid, application_uid, config)	OperatorNode class represents a non-model, operator node.
`PromptEvaluator`(*args, **kwargs)	An evaluator that uses LLM prompts to assess model outputs.

`Slice`(dataset, slice_uid, name[, ...])	Represents a slice within a Snorkel dataset for identifying and managing subsets of datapoints.

snorkelai.sdk.develop.Batch

class snorkelai.sdk.develop.Batch(*name, uid, dataset_uid, label_schemas, batch_size, ts, x_uids*)

Bases: `object`

The Batch object represents an annotation batch in Snorkel Flow. Currently, this interface only represents Dataset-level (not Node-level) annotation batches.

__init__

__init__(*name, uid, dataset_uid, label_schemas, batch_size, ts, x_uids*)

Create a batch object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods.

Parameters

Name	Type	Default	Info
name	`str`		The name of the batch.
uid	`int`		The UID for the batch within Snorkel Flow.
dataset_uid	`int`		The UID for the dataset within Snorkel Flow.
label_schemas	`List[LabelSchema]`		The list of label schemas associated with this batch.
batch_size	`int`		The number of examples in the batch.
ts	`datetime`		The timestamp at which the batch was created.
x_uids	`List[str]`		The UIDs for the examples in the batch.

Methods

`__init__`(name, uid, dataset_uid, ...)	Create a batch object in-memory with necessary properties.
`commit`(source_uid[, label_schema_uids])	Commit a source on a batch as ground truth.
`create`(dataset_uid[, name, assignees, ...])	Create one or more annotation batches for a dataset.
`delete`(batch_uid)	Delete an annotation batch by its UID.
`export`(path[, selected_fields, ...])	Export the batch to a zipped CSV file.
`get`(batch_uid)	Retrieve an annotation batch by its UID.
`get_dataframe`([selected_fields, ...])	Get a pandas DataFrame representation of the batch.
`update`([name, assignees, expert_source_uid])	Update properties of the annotation batch.

Attributes

`batch_size`	The number of examples in the batch.
`dataset_uid`	The UID for the dataset within Snorkel Flow.
`label_schemas`	The list of label schemas associated with this batch.
`name`	The name of the batch.
`ts`	The timestamp at which the batch was created.
`uid`	The UID for the batch within Snorkel Flow.
`x_uids`	The UIDs for the examples in the batch.

commit

commit(*source_uid, label_schema_uids=None*)

Commit a source on a batch as ground truth.

Parameters

Name	Type	Default	Info
source_uid	`int`		The UID for the source on the batch.
label_schema_uids	`Optional[List[int]]`	`None`	The label schema UIDs to commit, defaults to all label schemas if not set.

Return type

`None`

create

classmethod **create**(*dataset_uid, name=None, assignees=None, label_schemas=None, batch_size=None, randomize=False, random_seed=123, selection_strategy=None, split=None, x_uids=None, filter_by_x_uids_not_in_batch=False, divide_x_uids_evenly_to_assignees=False*)

Create one or more annotation batches for a dataset.

Typically, Dataset.create_batches() is the recommended entrypoint for creating batches.

Parameters

Name	Type	Defaul
dataset_uid	`int`	
name	`Optional[str]`	`None`

assignees	`Optional[List[int]]`	`None`
label_schemas	`Optional[List[LabelSchema]]`	`None`
batch_size	`Optional[int]`	`None`
num_batches		
randomize	`Optional[bool]`	`False`
random_seed	`Optional[int]`	`123`
selection_strategy	`Optional[SelectionStrategy]`	`None`
split	`Optional[str]`	`None`
x_uids	`Optional[List[str]]`	`None`

filter_by_x_uids_not_in_batch	`Optional[bool]`	`False`
divide_x_uids_evenly_to_assignees	`Optional[bool]`	`False`

Returns

The list of created batches

Return type

List[Batch]

delete

classmethod **delete**(*batch_uid*)

Delete an annotation batch by its UID.

Parameters

Name	Type	Default	Info
batch_uid	`int`		The UID for the batch within Snorkel Flow.

Return type

`None`

export

export(*path, selected_fields=None, include_annotations=False, include_ground_truth=False, max_rows=10000, csv_delimiter=',', quote_char='"', escape_char='\\\\'*)

Export the batch to a zipped CSV file.

Parameters

Name	Type	Default	Info
path	`Union[str, Path]`		The path to the zipped CSV file. If the path does not end in .zip, it will be appended to the path.
selected_fields	`Optional[List[str]]`	`None`	A list of fields to export. If not set, all fields will be exported.
include_annotations	`bool`	`False`	Whether to include annotations in the export.
include_ground_truth	`bool`	`False`	Whether to include ground truth in the export.
max_rows	`int`	`10000`	The maximum number of rows to export.

csv_delimiter	`str`	`','`	The delimiter to use for CSV fields.
quote_char	`str`	`'"'`	The character to use for quoted fields in the CSV.
escape_char	`str`	`'\\\\'`	The character to use for escaping special characters in the CSV.

Returns

The path to the zipped CSV file

Return type

`pathlib.Path`

get

classmethod **get**(*batch_uid*)

Retrieve an annotation batch by its UID.

Parameters

Name	Type	Default	Info
batch_uid	`int`		The UID for the batch within Snorkel Flow.

Returns

The batch object

Return type

Batch

get_dataframe

get_dataframe(*selected_fields=None, include_annotations=False, include_ground_truth=False, max_rows=10000*)

Get a pandas DataFrame representation of the batch.

Parameters

Name	Type	Default	Info
selected_fields	`Optional[List[str]]`	`None`	A list of fields to include in the DataFrame. If not set, all fields will be included.
include_annotations	`bool`	`False`	Whether to include annotations in the DataFrame.
include_ground_truth	`bool`	`False`	Whether to include ground truth in the DataFrame.
max_rows	`int`	`10000`	The maximum number of rows to include in the DataFrame.

Returns

The pandas DataFrame representation of the batch

Return type

`pd.DataFrame`

update

update(*name=None, assignees=None, expert_source_uid=None*)

Update properties of the annotation batch.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name of the batch.
assignees	`Optional[List[int]]`	`None`	The user UIDs for the new assignees of the batches.
expert_source_uid	`Optional[int]`	`None`	The UID for the new expert source of the batches.

Return type

`None`

property **batch_size***: int*

The number of examples in the batch.

property **dataset_uid***: int*

The UID for the dataset within Snorkel Flow.

property **label_schemas***: List[LabelSchema]*

The list of label schemas associated with this batch.

property **name***: str*

The name of the batch.

property **ts***: datetime*

The timestamp at which the batch was created.

property **uid**: *int*

The UID for the batch within Snorkel Flow.

property **x_uids**: *List[str]*

The UIDs for the examples in the batch.

snorkelai.sdk.develop.Benchmark

class snorkelai.sdk.develop.Benchmark(**args*, ***kwargs*)

Bases: `BaseModel`

A benchmark is the collection of characteristics that you care about for a particular GenAI application, and the measurements you use to assess the performance against those characteristics. It consists of the following elements:

- **Reference prompts:** A set of prompts used to evaluate the model's responses.
- **Slices:** Subsets of reference prompts focusing on specific topics.
- **Criteria:** Key characteristics that represent the features being optimized for evaluation.
- **Evaluators:** Functions that assess whether a model's output satisfies the criteria.

Read more in the Evaluation overview.

Using the `Benchmark` class requires the following import:

```
from snorkelai.sdk.develop import Benchmark
```

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark from which you want to get data. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
name	`str`		The name of the benchmark.
description	`Optional[str]`	`None`	The description of the benchmark.

created_at	`datetime`		The timestamp when the benchmark was created.
updated_at	`datetime`		The timestamp when the benchmark was last updated.
archived	`bool`		Whether the benchmark is archived.

__init__

__init__(*args, **kwargs)

Methods

`__init__`(*args, **kwargs)	
`archive`()	Archives the benchmark, hiding it from the UI and SDK list method.
`create`(name, dataset_uid[, description])	Creates a new benchmark.
`execute`([splits, criteria_uids, name])	Executes the benchmark against the associated dataset.
`export_config`(filepath[, format])	Exports a benchmark configuration to the specified format and writes to the provided filepath.
`export_latest_execution`(filepath[, config])	Export the latest benchmark execution with all its associated data.
`get`(benchmark_uid)	Gets a benchmark by its unique identifier.
`list`(workspace_uid[, include_archived])	Lists all benchmarks for a given workspace.
`list_criteria`([include_archived])	Retrieves all criteria for this benchmark.
`list_executions`([include_archived])	Retrieves all benchmark executions for this benchmark.
`update`([name, description, archived])	Updates the benchmark with the given parameters.

Attributes

`description`	
`benchmark_uid`	
`name`	
`created_at`	
`updated_at`	
`archived`	

archive

archive()

Archives the benchmark, hiding it from the UI and SDK list method.

Use `snorkelai.sdk.develop.benchmarks.Benchmark.list()` with `include_archived=True` to view archived benchmarks.

Return type

`None`

create

static **create**(*name, dataset_uid, description=None*)

Creates a new benchmark. The created benchmark does not include any default criteria or evaluators.

Parameters

Name	Type	Default	Info
name	`str`		The name of the benchmark.
dataset_uid	`int`		The unique identifier of the dataset t the benchmark. The `dataset_uid` c the

			`snorkelai.sdk.develop.datase` method.
description	`Optional[str]`	`None`	The description of the benchmark.

Returns

A Benchmark object representing the created benchmark.

Return type

Benchmark

execute

execute(*splits=None, criteria_uids=None, name=None*)

Executes the benchmark against the associated dataset. For each criteria, evaluation scores are computed for each datapoint and aggregate metrics are computed across all datapoints.

Parameters

Name	Type	Default	Info
splits	`Optional[List[str]]`	`None`	The splits to execute the benchmark on. If not provided, will default to [“train”, “valid”].
criteria_uids	`Optional[List[int]]`	`None`	The criteria to execute the benchmark on. If not provided, will default to all criteria for the benchmark.
name	`Optional[str]`	`None`	The name of the execution. If not provided, will default to “Run <number>” based on the

			number of previous executions.

Returns

The execution object.

Return type

BenchmarkExecution

export_config

export_config(*filepath, format=BenchmarkExportFormat.JSON*)

Exports a benchmark configuration to the specified format and writes to the provided filepath.

This method exports the complete benchmark configuration, including all criteria, evaluators, and metadata. The exported configuration can be used for:

- Version control of benchmark definitions.
- Sharing benchmarks across teams.
- Integration with CI/CD pipelines.
- Backing up evaluation configurations.

Parameters

Name	Type	Default

filepath	`str`		C p e d d w c it e
format	`BenchmarkExportFormat`	`<BenchmarkExportFormat.JSON: 'json'>`	T f e c C o is s

Raises

- **NotImplementedError** – If an unsupported export format is specified.
- **ValueError** – If the benchmark_uid is None or invalid.

Return type

`None`

Example

Example 1

Export a benchmark configuration to JSON:

```
benchmark = Benchmark.get(100)
benchmark.export_config("benchmark_config.json")
```

Example 1 output

The exported JSON file contains:

```
{
  "criteria": [
    {
      "criteria_uid": 101,
      "benchmark_uid": 100,
      "name": "Example Readability",
      "description": "Evaluates how easy the response is to read
and understand.",
      "state": "ACTIVE",
      "output_format": {
        "metric_label_schema_uid": 200,
        "rationale_label_schema_uid": 201
      },
      "metadata": {
        "version": "1.0"
      },
      "created_at": "2025-04-01T14:30:00.123456Z",
      "updated_at": "2025-04-01T14:35:10.654321Z"
    }
  ],
  "evaluators": [
    {
      "evaluator_uid": 301,
      "name": "Readability Evaluator (LLM)",
      "description": "Uses an LLM prompt to assess readability.",
      "criteria_uid": 101,
      "type": "Prompt",
      "prompt_workflow_uid": 401,
      "parameters": null,
      "metadata": {
        "default_prompt_config": {
          "name": "Readability Prompt v1",
          "model_name": "google/gemini-1.5-pro-latest",
          "system_prompt": "You are an expert evaluator assessing
text readability.",
          "user_prompt": "..."
        }
      },
      "created_at": "2025-04-01T15:00:00.987654Z",
      "updated_at": "2025-04-01T15:05:00.123123Z"
    }
  ],
  "metadata": {
    "name": "Sample Benchmark Set",
    "description": "A benchmark set including example
evaluations.",
    "created_at": "2025-04-01T14:00:00.000000Z",
    "created_by": "user@example.com"
    }
                }
```

After exporting your benchmark, you can use it to evaluate data from your GenAI application iteratively, allowing you to measure and refine your LLM system.

export_latest_execution

export_latest_execution(*filepath, config=None*)

Export the latest benchmark execution with all its associated data.

This method exports the most recent benchmark execution, including all evaluation results and metadata. The exported dataset contains:

- Benchmark metadata for the associated benchmark
- Execution metadata for this execution
- Each datapoint lists its evaluation score, which includes:
  - The evaluator outputs
  - Rationale
  - Agreement with ground truth
- Each datapoint lists its slice membership(s)
- (CSV exports only) Uploaded user columns and ground truth

The export includes all datapoints without filtering or sampling. Some datapoints may have missing evaluation scores if the benchmark was not executed against them (for example, datapoints in the test split).

Parameters

Name	Type	Default	Info
filepath	`str`		Output file path for exported data.
config	`Union[JsonExportConfig, CsvExportConfig, None]`	`None`	A `JsonExportConfig` or `CsvExportConfig` object. If not provided, JSON will be used by

			default. No additional configuration is required for JSON exports. For CSV exports, the following parameters are supported: • `sep`: The separator between columns. Default: `,`. • `quotechar`: The character used to quote fields. Default: `"`. • `escapechar`: The character used to escape special characters. Default: `\`.

Return type

`None`

Example

Example 1

Export the latest benchmark execution to JSON:

```
benchmark = Benchmark.get(100)
benchmark.export_latest_execution("benchmark_execution.json")
```

Example 1 return

The exported JSON file contains:

```
{
    "benchmark_metadata": {
        "uid": 100,
        "name": "Example Benchmark",
        "description": "A benchmark for testing model performance",
        "created_at": "2025-01-01T12:00:00Z",
        "created_by": "user@example.com"
    },
    "execution_metadata": {
        "uid": 1,
        "name": "Latest Run",
        "created_at": "2025-01-01T12:00:00Z",
        "created_by": "user@example.com"
    },
    "data": [
        {
            "x_uid": "doc::0",
            "scores": [
               {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "RATIONALE",
                    "value": "The response is clear and well-
structured",
                    "error": ""
               },
               {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "EVAL",
                    "value": 0.85,
                },
                {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "AGREEMENT",
                    "value": 1.0
                }
            ],
            "slice_membership": ["test_set"]
        },
        {
            "x_uid": "doc::1",
            "scores": [
                {
                    "criteria_uid": 101,
                    "criteria_name": "Readability",
                    "score_type": "EVAL",
                    "value": 0.92,
                }
            ],
            "slice_membership": ["test_set"]
        }
    ],
    "slices": [
        {
            "id": "None",
            "display_name": "All Datapoints",
            "reserved_slice_type": "global"
        },
        {
            "id": "-1",
            "display_name": "No Slice",
            "reserved_slice_type": "no_slice"
        },
        {
            "id": "5",
            "display_name": "Your Slice",
            "reserved_slice_type": "regular_slice"
        }
    ]
}
```

get

static **get**(*benchmark_uid*)

Gets a benchmark by its unique identifier.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark from which you want to get data. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.

Returns

A Benchmark object representing the benchmark with the given `benchmark_uid`.

Return type

Benchmark

list

static **list**(*workspace_uid, include_archived=False*)

Lists all benchmarks for a given workspace.

Parameters

Name	Type	Default	Info
workspace_uid	`int`		The unique identifier of the workspace fron list benchmarks. The `workspace_uid` can the **`snorkelai.sdk.client_v3.utils.get`** method.
include_archived	`bool`	`False`	Whether to include archived benchmarks.

Returns

A list of Benchmark objects representing all benchmarks in the given workspace.

Return type

List[Benchmark]

list_criteria

list_criteria(*include_archived=False*)

Retrieves all criteria for this benchmark.

Criteria are the key characteristics that represent the features being optimized for evaluation. Each criteria defines what aspect of the model's performance is being measured, such as accuracy, relevance, or safety.

Each Criteria object contains:

- criteria_uid: The unique identifier for this criteria.
- benchmark_uid: The ID of the parent benchmark.
- name: The name of the criteria.
- description: A detailed description of what the criteria measures.
- requires_rationale: Whether the criteria requires a rationale explanation.
- label_map: A dictionary mapping user-friendly labels to numeric values.

Parameters

Name	Type	Default	Info
include_archived	`bool`	`False`	Whether to include archived criteria.

Returns

A list of Criteria objects representing all criteria in this benchmark.

Return type

List[Criteria]

Example

Example 1

Get all criteria for a benchmark and list them:

```
benchmark = Benchmark.get(100)
criteria_list = benchmark.list_criteria()
for criteria in criteria_list:
    print(f"Criteria: {criteria.name} - {criteria.description}")
```

list_executions

list_executions(*include_archived=False*)

Retrieves all benchmark executions for this benchmark.

A benchmark execution represents a single run of a benchmark against a dataset, capturing the results and metadata of that evaluation. Executions are returned in chronological order, with the most recent execution last.

Each BenchmarkExecution object contains:

- benchmark_uid: The ID of the parent benchmark.
- benchmark_execution_uid: The unique identifier for this execution.
- name: The name of the execution.
- created_at: Timestamp when the execution was created.
- created_by: Username of the execution creator.
- archived: Whether the execution is archived.

After retrieving executions, you can export their results using `export_latest_execution()` or export the benchmark configuration using `export_config()`. For more information about exporting benchmarks, see Export evaluation benchmark.

Parameters

Name	Type	Default	Info
include_archived	`bool`	`False`	Whether to include archived executions.

Return type

List[BenchmarkExecution]

Example

Example 1

Get all executions for a benchmark and list them:

```
benchmark = Benchmark.get(100)
executions = benchmark.list_executions()
```

update

update(*name=None, description=None, archived=None*)

Updates the benchmark with the given parameters. If a parameter is not provided or is None, the existing value will be left unchanged.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name of the benchmark.
description	`Optional[str]`	`None`	The new description of the benchmark.
archived	`Optional[bool]`	`None`	Whether the benchmark should be archived.

Returns

A Benchmark object representing the updated benchmark.

Return type

Benchmark

Example

```
benchmark = Benchmark.get(100)
benchmark.update(name="New Name", description="New description")
```

archived: `bool`

benchmark_uid: `int`

created_at: `datetime`

description: `Optional` [`str`] = None

name: `str`

updated_at: `datetime`

snorkelai.sdk.develop.BenchmarkExecution

class snorkelai.sdk.develop.BenchmarkExecution(**args, **kwargs*)

Bases: `BaseModel`

Represents a single execution run of a benchmark for a dataset.

A benchmark execution exports comprehensive evaluation data including per-datapoint scores (evaluator outputs, rationales, and ground truth agreement), slice membership, benchmark and execution metadata, including timing information and execution context.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the parent Benchmark. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
benchmark_execution_uid	`int`		The unique identifier for this execution.
name	`str`		The name of the execution.
created_at	`datetime`		Timestamp of when this execution was run.
created_by	`str`		Username of the user who ran this execution.
archived	`bool`		Whether this execution is archived.

__init__

__init__(**args, **kwargs*)

Methods

`__init__`(*args, **kwargs)	
`export`(filepath[, config, connector_config_uid])	Export information associated with this benchmark execution.
`list`(benchmark_uid[, include_archived])	List all benchmark executions for a given benchmark.
`update`(archived)	Update the state of the benchmark execution.

Attributes

`benchmark_uid`	
`benchmark_execution_uid`	
`name`	
`created_at`	
`created_by`	
`archived`	

export

export(*filepath, config=None, connector_config_uid=None*)

Export information associated with this benchmark execution. The exported data includes:

- Benchmark metadata for the associated benchmark
- Execution metadata for this execution
- Each datapoint lists its evaluation score, which includes:
    - The evaluator outputs
    - Rationale

  - Agreement with ground truth
- Each datapoint lists its slice membership(s)
- (CSV exports only) Uploaded user columns and ground truth

The export includes all datapoints without filtering or sampling. Some datapoints may have missing evaluation scores if the benchmark was not executed against them (for example, datapoints in the test split).

Parameters

Name	Type	Default	Info
filepath	`str`		The filepath wl you want to wr exported data.
config	`Union[JsonExportConfig, CsvExportConfig, None]`	`None`	A `JsonExportC` or `CsvExportCo` object. Default JSON. No addit configuration i required for JS exports. For CS exports, the following para are supported: • `sep`: The separator between co Default is `,` • `quotechar` character u quote fields Default is `"`

			• `escapecha` The charact used to esc special characters. Default is `\`
connector_config_uid	`Optional[int]`	`None`	Optional UID c connector con use for the exp **Required** only export destina a remote, priv bucket (a priva or GCS bucket requires crede **Ignored** if the destination is public bucket (public S3 or G bucket that do require creder or if the expor destination is file.

Return type

`None`

Examples

Example 1

Export a benchmark execution to a local file:

```
from snorkelai.sdk.develop import Benchmark

benchmark = Benchmark.get(100)
execution = benchmark.list_executions()[0]
execution.export("benchmark_execution.json")
```

Example 2

Export a benchmark execution to a S3 bucket using a connector config:

```
from snorkelai.sdk.develop import Benchmark

benchmark = Benchmark.get(100)
execution = benchmark.list_executions()[0]
execution.export("s3://MY-BUCKET/MY-PATH/benchmark_execution.json",
connector_config_uid=1)
```

list

static **list**(*benchmark_uid, include_archived=False*)

List all benchmark executions for a given benchmark.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the parent Benchmark. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
include_archived	`bool`	`False`	Whether to include archived executions. Defaults to False.

Return type

`List[BenchmarkExecution]`

update

update(*archived*)

Update the state of the benchmark execution.

Parameters

Name	Type	Default	Info
archived	`bool`		Whether the benchmark execution should be archived.

Return type

`None`

archived: *bool*

benchmark_execution_uid: *int*

benchmark_uid: *int*

created_at: *datetime*

created_by: *str*

name: *str*

snorkelai.sdk.develop.CodeEvaluator

class snorkelai.sdk.develop.CodeEvaluator(**args*, ***kwargs*)

Bases: `Evaluator`

An evaluator that uses custom Python code to assess an AI application's responses.

A code evaluator uses custom Python functions to evaluate datapoints containing AI application responses, categorizing them into one of a criteria's labels by assigning the corresponding integer score and optional rationale. The evaluator function takes a datapoint as input and returns a score based on the criteria's label schema.

The evaluation function can implement any Python logic needed to assess the AI application's response.

Read more in the Evaluation overview.

Using the `CodeEvaluator` class requires the following import:

```
from snorkelai.sdk.develop import CodeEvaluator
```

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark that contains the criteria. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
evaluator_uid	`int`		The unique identifier for this evaluator.

Examples

Example 1

Creates a new code evaluator, assessing the length of the AI application’s response:

```
import pandas as pd

def evaluate(df: pd.DataFrame) -> pd.DataFrame:
    results = pd.DataFrame(index=df.index)
    results["score"] = df["response"].str.len() > 10  # Simple length
check
    results["rationale"] = "Response length evaluation"
    return results

# Create a new code evaluator
evaluator = CodeEvaluator.create(
    criteria_uid=100,
    evaluate_function=evaluate,
    version_name="Version 1"
)
```

Example 2

Gets an existing code evaluator:

```
# Get existing evaluator
evaluator = CodeEvaluator.get(
    evaluator_uid=300,
)
```

__init__

__init__(**args, **kwargs*)

Methods

`__init__`(*args, **kwargs)	
`create`(criteria_uid, **kwargs)	Creates a new code evaluator for a criteria.
`execute`(split[, num_rows, version_name, sync])	Executes the code evaluator against a dataset split.
`get`(evaluator_uid)	Retrieves a code evaluator for a given uid.

`get_execution_result`(execution_uid)	Retrieves the evaluation results for a specific evaluation execution.
`get_executions`()	Retrieves all executions for this code evaluator.
`get_versions`()	Retrieves all code version names for this code evaluator.
`poll_execution_result`(execution_uid[, sync])	Polls the job status and retrieves partial results.
`update`([version_name])	Updates the code evaluator with a new evaluation function.

Attributes

`benchmark_uid`	
`criteria_uid`	
`evaluator_uid`	

create

classmethod **create**(*criteria_uid, **kwargs*)

Creates a new code evaluator for a criteria.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
**kwargs	`Any`		Additional parameters. Must include: <ul><li>evaluate_function : Callable[[pd.DataFrame], pd.DataFrame] A Python function that performs the evaluation. This function must:<ul><li>Be named `evaluate`</li><li>Accept a pandas DataFrame as input</li></ul></li></ul>

  - Return a pandas DataFrame as output

  The input DataFrame has a MultiIndex with a single level named `__DATAPOINT_UID` that holds the unique identifier of the datapoint. Values in the index are of the form `("uid1",)`.

  The output DataFrame must:

  - Have the same index as the input DataFrame
  - Include a column named `score` containing the evaluation results
  - Optionally include a column named `rationale` with explanations for the scores

May include:

- version_name : str The name for the initial code version. If not provided, a default name will be generated.

Raises

ValueError – If the function name is not `evaluate` or if `evaluate_function` is not callable.

Return type

`CodeEvaluator`

Example

Example 1

Creates a new code evaluator, assessing the length of the AI application’s response:

```
import pandas as pd

def evaluate(df: pd.DataFrame) -> pd.DataFrame:
    results = pd.DataFrame(index=df.index)
    results["score"] = df["response"].str.len() > 10
    results["rationale"] = "Response length evaluation"
    return results

evaluator = CodeEvaluator.create(
    criteria_uid=100,
    evaluate_function=evaluate,
)
```

execute

execute(*split, num_rows=None, version_name=None, sync=False, **kwargs*)

Executes the code evaluator against a dataset split.

This method runs the evaluation code against the specified dataset split. If no version name is specified, it uses the latest version.

Parameters

Name	Type	Default	Info
split	`str`		The dataset split to evaluate against (e.g., “train”, “test”, “validation”).
num_rows	`Optional[int]`	`None`	The number of rows to evaluate. If `None`, evaluates all rows in the split.
version_name	`Optional[str]`	`None`	The code version name to run. If `None`, the latest code version is used.
sync	`bool`	`False`	Whether to wait for the job to complete. If `True`, blocks until completion.

Return type

`int`

Example

Example 1

Run the latest code version and poll for results:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(split="test", num_rows=100)
status, results =
evaluator.poll_execution_result(code_execution_uid, sync=False)
```

Example 2

Run a specific code version:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(
    split="test",
    num_rows=100,
    version_name="v1.0"
)
```

get

classmethod **get**(*evaluator_uid*)

Retrieves a code evaluator for a given uid.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier for the evaluator.

Returns

A CodeEvaluator instance.

Return type

CodeEvaluator

Raises

ValueError – If the evaluator with the given uid is not a CodeEvaluator.

get_execution_result

get_execution_result(*execution_uid*)

Retrieves the evaluation results for a specific evaluation execution.

This method reads the evaluation results from the database for the given evaluation execution UID.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The evaluation execution UID to get results for.

Return type

`Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]

Example

Example 1

Get the results of a code execution:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(split="test", num_rows=100)
results = evaluator.get_execution_result(code_execution_uid)
print(f"Evaluation scores: {results}")
```

get_executions

get_executions()

Retrieves all executions for this code evaluator.

This method fetches all executions that have been run using this evaluator. Executions are returned in chronological order, with the oldest execution first.

The dictionary contains the following keys: :rtype: `List`[`Dict`[`str`, `Any`]]

- `execution_uid`: The execution UID
- `created_at`: The timestamp when the execution was created
- `code_version_name`: The name of the code version used for the execution

Example

Example 1

Get all executions for an evaluator:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
executions = evaluator.get_executions()
for execution in executions:
    print(f"Execution {execution['execution_uid']}:
{execution['created_at']}")
```

get_versions

get_versions()

Retrieves all code version names for this code evaluator.

This method fetches all code version names that have been created for this evaluator. Versions are returned in chronological order, with the oldest version first.

Return type

`List[str]`

Example

Example 1

Get all code version names for an evaluator:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
versions = evaluator.get_versions()
for version in versions:
    print(f"Version: {version}")
```

poll_execution_result

poll_execution_result(*execution_uid, sync=False*)

Polls the job status and retrieves partial results.

This method checks the current status of the evaluation job and returns both the job status and any available results. The current status can be `running`, `completed`, `failed`, `cancelled`, or `unknown`.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The code execution UID to poll for.
sync	`bool`	`False`	Whether to wait for the job to complete. If `False`, returns immediately with current status and partial results.

Return type

`Tuple[str, Dict[str, Dict[str, Union[str, int, float, bool]]]]`

Example

Example 1

Poll for job status and partial results:

```
evaluator = CodeEvaluator.get(evaluator_uid=300)
code_execution_uid = evaluator.execute(split="test", num_rows=100)
status, results =
evaluator.poll_execution_result(code_execution_uid, sync=False)
print(f"Job status: {status}")
if results:
    print(f"Partial results: {results}")
```

update

update(*version_name=None, **kwargs*)

Updates the code evaluator with a new evaluation function.

This method creates a new code version containing the provided evaluation function. The function must be declared with the name `evaluate` and will be used to assess datapoints containing AI application responses against the criteria.

Parameters

Name	Type	Default	Info
version_name	`Optional[str]`	`None`	The name for the new code version. If not provided, a default name will be generated.
**kwargs	`Any`		Additional parameters. Must include: • evaluate_function : Callable[[pd.DataFrame], pd.DataFrame] A Python function that performs the evaluation. This function must: ◦ Be named `evaluate`

			◦ Accept a pandas DataFrame as input ◦ Return a pandas DataFrame as output The input DataFrame has a MultiIndex with a single level named `__DATAPOINT_UID` that holds the unique identifier of the datapoint. Values in the index are of the form `("uid1",)`. The output DataFrame must: ◦ Have the same index as the input DataFrame ◦ Include a column named `score` containing the evaluation results ◦ Optionally include a column named `rationale` with explanations for the scores

Raises

ValueError – If the function name is not ‘evaluate’ or if evaluate_function is not provided.

Return type

`str`

Example

Example 1

Update a code evaluator with a new evaluation function:

```python
import pandas as pd

def evaluate(df: pd.DataFrame) -> pd.DataFrame:
    results = pd.DataFrame(index=df.index)

    # Add random scores between 0 and 1
    results["score"] = np.random.randint(0, 3, size=len(df))

    # Add random rationales
    rationale_options = [
        "This response is accurate and relevant.",
        "The answer demonstrates good understanding.",
        "Response shows appropriate reasoning.",
        "This is a well-formed answer.",
        "The content is factually correct.",
    ]
    results["rationale"] = np.random.choice(rationale_options,
size=len(df))
    return results

evaluator = CodeEvaluator.get(evaluator_uid=300)
version_name = evaluator.update("v2.0", evaluate_function=evaluate)
```

benchmark_uid: `int`

criteria_uid: `int`

evaluator_uid: `int`

snorkelai.sdk.develop.Criteria

class snorkelai.sdk.develop.Criteria(**args*, ***kwargs*)

Bases: `BaseModel`

A criteria represents a specific characteristic or feature being evaluated as part of a benchmark.

Criteria define what aspects of a model or AI application's performance are being measured, such as accuracy, relevance, safety, and other qualities. Each criteria is associated with a benchmark and has an evaluator that assesses whether a model's output satisfies that criteria.

The heart of each criteria is its associated label schema, which defines what, exactly, the criteria is measuring, and maps each option to an integer.

For example, a criteria that measures accuracy might have a label schema that defines the following labels:

- `INCORRECT`: 0
- `CORRECT`: 1

A criteria that measures readability might have a label schema that defines the following labels:

- `POOR`: 0
- `ACCEPTABLE`: 1
- `EXCELLENT`: 2

Read more in the Evaluation overview.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of th parent Benchmark. The `benchmark_uid` is visible the URL of the benchmark page in the Snorkel GUI. Fo example, `https://YOUR-`

			`SNORKEL-INSTANCE/benchmarks/` indicates a benchmark wit `benchmark_uid` of `100`.
criteria_uid	`int`		The unique identifier for tl criteria.
name	`str`		The name of the criteria.
metric_label_schema_uid	`int`		The ID of the schema defir the metric labels.
description	`str`	`""`	A detailed description of w the criteria measures.
rationale_label_schema_uid	`Optional[int], default=None`		The ID of the schema defir rationale labels (if applicat

Examples

Using the `Criteria` class requires the following import:

```
from snorkelai.sdk.develop import Criteria
```

Create a new criteria:

```
# Create a new criteria
criteria = Criteria.create(
    benchmark_uid=100,
    name="Accuracy",
    description="Measures response accuracy",
    label_map={"Correct": 1, "Incorrect": 0},
    requires_rationale=True
)
```

Get an existing criteria:

```
# Get existing criteria
criteria = Criteria.get(criteria_uid=100)
```

__init__

__init__(*args, **kwargs)

Methods

`__init__`(*args, **kwargs)	
`archive`()	Archives the criteria, hiding it from the UI and Benchmark.list_criteria method.
`create`(benchmark_uid, name, label_map [, ...])	Create a new criteria for a benchmark.
`get`(criteria_uid)	Get an existing criteria by its UID.
`get_evaluator`()	Retrieves the evaluator associated with this criteria.
`update`([name, description, archived])	Updates the criteria with the given parameters.

Attributes

`archived`	
`description`	
`rationale_label_schema_uid`	
`benchmark_uid`	
`criteria_uid`	
`name`	
`metric_label_schema_uid`	

archive

archive()

Archives the criteria, hiding it from the UI and Benchmark.list_criteria method.

Use `snorkelai.sdk.develop.benchmarks.Benchmark.list_criteria()` with `include_archived=True` to view archived criteria.

Return type

`None`

create

static **create**(*benchmark_uid, name, label_map, description=None, requires_rationale=False*)

Create a new criteria for a benchmark.

Your `label_map` must use consecutive integers starting from `0`. For example, if you have three labels, you must use the values `0`, `1`, and `2`.

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the parent Benchmark.
name	`str`		The name of the criteria.
label_map	`Dict[str, int]`		A dictionary mapping user-friendly labels to numeric values. The key “UNKNOWN” will always be added with value -1. Dictionary values must be consecutive integers starting from 0.
description	`str, default=None`		A detailed description of what the criteria measures.
requires_rationale	`bool, default=False`		Whether the criteria requires rationale.

Returns

A new Criteria object representing the created criteria.

Return type

Criteria

Raises

ValueError – If label_map is empty or has invalid values.

Example

```
criteria = Criteria.create(
    benchmark_uid=200,
    name="Accuracy",
    description="Measures response accuracy",
    label_map={"Correct": 1, "Incorrect": 0},
    requires_rationale=True
)
```

get

static **get**(*criteria_uid*)

Get an existing criteria by its UID.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier for the criteria.

Returns

A Criteria object representing the existing criteria.

Return type

Criteria

Raises

ValueError – If the criteria is not found.

Example

```
criteria = Criteria.get(criteria_uid=100)
```

get_evaluator

get_evaluator()

Retrieves the evaluator associated with this criteria.

An evaluator is a prompt or code snippet that assesses whether a model’s output satisfies the criteria. Each criteria has one evaluator that assesses each datapoint against the criteria’s label schema and chooses the most appropriate label, in the form of the associated integer.

The evaluator can be either a code evaluator (using custom Python functions) or a prompt evaluator (using LLM prompts).

Raises

IndexError – If no evaluator is found for this criteria.

Return type

`Evaluator`

Example

Example 1

Get the evaluator for a criteria and check its type:

```
from snorkelai.sdk.develop import PromptEvaluator, CodeEvaluator

criteria = Criteria.get(criteria_uid=100)
evaluator = criteria.get_evaluator()

if isinstance(evaluator, CodeEvaluator):
    print("This is a code evaluator")
elif isinstance(evaluator, PromptEvaluator):
    print("This is a prompt evaluator")
```

update

update(*name=None, description=None, archived=None*)

Updates the criteria with the given parameters. If a parameter is not provided or is None, the existing value will be left unchanged.

Parameters

Name	Type	Default	Info
name	`str, default=None`		The name of the criteria.
description	`str, default=None`		A detailed description of what the criteria measures.
archived	`bool, default=None`		Whether the criteria is archived.

Returns

The updated Criteria object.

Return type

Criteria

Example

```
criteria = Criteria.get(criteria_uid=100)
criteria.update(name="New Name", description="New description")
```

archived: `bool` = False

benchmark_uid: `int`

criteria_uid: `int`

description: `Optional`[`str`] = None

metric_label_schema_uid: `int`

name: `str`

rationale_label_schema_uid: `Optional`[`int`] = None

snorkelai.sdk.develop.CsvExportConfig

class snorkelai.sdk.develop.CsvExportConfig(*sep=',', quotechar='"', escapechar='\\\\'*)

Bases: `object`

__init__

__init__(*sep=',', quotechar='"', escapechar='\\\\'*)

Methods

`__init__`([sep, quotechar, escapechar])	

snorkelai.sdk.develop.Dataset

class snorkelai.sdk.develop.Dataset(*name, uid, mta_enabled*)

Bases: `object`

The Dataset object represents a dataset in Snorkel Flow.

Datasets Quickstart

In this quickstart, we will create a Dataset and upload a file to that Dataset as a data source. We will then show how you might go about ingesting that data into the platform.

We will need the following imports

```
from snorkelai.sdk.develop import Dataset
import snorkelai.sdk.client as sf
import pandas as pd
ctx = sf.SnorkelSDKContext.from_endpoint_url()
```

We will begin by creating a new Dataset.

```
>>> contracts_dataset = Dataset.create("contracts-dataset")
Successfully created dataset contracts-dataset with UID 0 in workspace 0
```

Next, we will attempt to save a file to the Dataset as a data source. This file will be in S3. File upload will initially fail because this file contains null values.

```
>>> contracts_dataset.create_datasource("s3://snorkel-contracts-
dataset/dev.parquet", uid_col="uid", split="train")
UserInputError: Errors...
```

In this particular example, we decide we don't care about these rows, so we can use Pandas to edit the file and remove the null values. We can then re-upload the data, this time uploading the DataFrame directly without needing to save it to a file again. In some other cases, you may want to either edit those null cells or fix them in your upstream data pipeline.

```
>>> df = pd.read_parquet("s3://snorkel-contracts-dataset/dev.parquet")
>>> df = df.dropna()
>>> contracts_dataset.create_datasource(df, uid_col="uid",
split="train")
+0.07s Starting data ingestion
+1.85s Ingesting data
+2.05s Data ingestion complete
```

To verify that has worked correctly, we can view this Dataset's data sources.

```
>>> contracts_dataset.datasources
[{'datasource_uid': 668,...}]
```

Dataset Concepts

Datasets

Datasets are how your data is represented in Snorkel Flow. Snorkel Flow projects always begin with a single Dataset. Datasets bring external data into Snorkel Flow and help manage that data once it has been ingested. Datasets are composed of individual chunks of data, called **data sources**, and provides an interface for managing individual data sources.

Data Sources

Data sources are the individual chunks of data that make up a Dataset. A data source can be a file you upload from local storage, a file located in a remote (S3, MinIO, etc.) storage service, or an in-memory Pandas DataFrame. Data sources shouldn't be touched directly, but should be managed by interfacing with their parent Dataset. The best way to deal with data sources is to treat them as blocks of data, which can be added and removed but only occasionally changed. Data sources can be given names during their creation, but are usually referred to using a data source UID, an integer ID assigned to each data source when it is created.

Derived Data Sources

When an application is created using a dataset, Snorkel Flow will create a *derived* data source for each data source in the dataset. Derived data sources are intermediate representations of data that track the lineage of the data as it is being processed and are associated with only one application. Note that some operations, such as changing the split of a data source, don't propagate to any of the derived data source once they are derived, and vice versa. Derived data sources are viewable in the Snorkel Flow UI on the "View Data Sources" button, accessible from the "Develop" screen of an application.

Modifying Data

In general, data sources should be treated as immutable. This means that you should avoid modifying the underlying data source once it has been uploaded. If your goal is to filter out rows, add feature columns, or remove feature columns, you should use an Operator to do so. Alternatively, you can modify your data upstream of Snorkel and create a new Dataset with your edited data.

The Python SDK provides limited support for specific one-off operations on data sources. Sometimes you might need to reformat the data in an existing column to make it compatible with processing logic. In this case, you can use the `dataset.update_datasource_data` method to swap out an existing data source for a new one with the updated data. However, be aware that this is an irreversible change, and updating data in this way is an expensive operation that will require all downstream applications to be refreshed.

Splits

Data sources belong to *splits*. Splits help dictate how the data will be used in the model development process. Data sources allocated to the **train** split will be used for model training and labeling function development. Data sources allocated to the **valid** split will be used to validate models iteratively and to perform error analysis. Data sources allocated to the **test** split will be used to evaluate the final model. Data source splits may be updated as needed, but be aware that model metrics and labeling function performance will change based on how the splits are allocated.

Data Upload Guardrails

When you upload data to Snorkel Flow, it must pass a series of safety checks to ensure that the data is valid and safe to load into the platform. These checks include:

> - **Number of rows**: A single data source should not exceed 10 million rows. If your data source exceeds this limit, you should split it into multiple data sources before uploading.
> - **Column memory**: The average memory usage of a single column must be under 20MB across all columns in your data source. For performance, the average column memory usage should be under 5MB. If your data source exceeds this limit, you should split it into multiple data sources before uploading.

- **Null values**: Snorkel Flow will not permit data to be uploaded if any null values exist in that data source. If you have null values in your data, you might want to clean them up with the Pandas `fillna()` method before uploading.
- **Unique integer index**: Snorkel Flow requires that each data source have a unique integer index column. The values in this index must be unique among all datasources in the Dataset. The values must also be unique, non-negative integers. If your Dataset does not already have this stable index column, you must create one before uploading.
- **Consistent schema**: All data sources in a single Dataset should have the same columns. All columns that are in multiple data sources must have the same type. If you have columns that exist in some data sources but not others, you may see unexpected behavior in downstream tasks.

Fetching UIDs

Methods in the `Dataset` class will sometimes require a UID parameter. This is the unique identifier for the Dataset within Snorkel Flow. The Dataset UID can be retrieved by calling `.uid` on a Dataset object. Data source methods will sometimes require a data source UID, which can be retrieved by printing out the datasources by calling `my_dataset.datasources`. The data source UID is the `datasource_uid` field in the returned dictionary.

__init__

__init__(*name, uid, mta_enabled*)

Create a dataset object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods

Parameters

Name	Type	Default	Info
name	`str`		The human-readable name of the dataset. Must be unique within the workspace.

uid	`int`		The unique integer identifier for the dataset within Snorkel Flow.
mta_enabled	`bool`		Whether or not multi-task annotation is enabled for this dataset.

Methods

`__init__`(name, uid, mta_enabled)	Create a dataset object in-memory with necessary properties.
`create`(dataset_name[, enable_mta])	Creates and registers a new Dataset object.
`create_batches`([name, assignees, ...])	Create annotation batches for this dataset.
`create_datasource`(data[, uid_col, name, ...])	Creates a new data source withing the Dataset from either a filepath or a Pandas DataFrame.
`create_label_schema`(name, data_type, ...[, ...])	Create a label schema associated with this dataset.
`delete`(dataset[, force])	Delete a dataset based on the provided identifier
`delete_datasource`(datasource_uid[, force, sync])	Delete a data source.
`get`(dataset)	Fetches an already-created Dataset from Snorkel Flow and returns a Dataset object that can be used to interact with files and data
`get_dataframe`([split, max_rows, ...])	Read the Dataset's data into an in-memory Pandas DataFrame.
`list`()	Get a list of all Datasets.
`update`([name])	Update the metadata for this dataset.
`update_datasource_data`(old_datasource_uid, ...)	This function allows you to replace the data of an existing data source with new data.
`update_datasource_split`(datasource_uid, split)	Change the split of a data source that has already been uploaded to the dataset.

Attributes

`batches`	A list of batches belonging to this Dataset.
`datasources`	A list of data sources and associated metadata belonging to this Dataset.
`label_schemas`	A list of label schemas belonging to this Dataset.
`mta_enabled`	Whether or not multi-task annotation is enabled for this dataset.
`name`	The human-readable name of the dataset.
`uid`	The unique integer identifier for the dataset within Snorkel Flow.

create

classmethod **create**(*dataset_name, enable_mta=True*)

Creates and registers a new Dataset object. A Dataset object organizes and collects files and other sources of data for use in Snorkel Flow. A Dataset is restricted to a particular workspace, so only users in that workspace will be able to access that Dataset. Datasets must be initialized with a unique name

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.create(dataset_name="my-dataset")
Successfully created dataset my-dataset with UID 0 in workspace 0
```

Parameters

Name	Type	Default	Info
dataset_name	`str`		A name for the Dataset. This name must be unique within the workspace.
enable_mta	`bool`	`True`	Whether to enable multi-task annotation for this dataset. Enabled by default.

Returns

A Dataset object that can be used to interact with the dataset in Snorkel Flow

Return type

Dataset

create_batches

create_batches(*name=None, assignees=None, label_schemas=None, batch_size=None, randomize=False, random_seed=123, selection_strategy=None, split=None, x_uids=None, filter_by_x_uids_not_in_batch=False, divide_x_uids_evenly_to_assignees=False*)

Create annotation batches for this dataset.

This is the recommended entrypoint for creating batches.

Parameters

Name	Type	Defaul
name	`Optional[str]`	`None`
assignees	`Optional[List[int]]`	`None`
label_schemas	`Optional[List[LabelSchema]]`	`None`
batch_size	`Optional[int]`	`None`
num_batches		

randomize	`Optional[bool]`	`False`
random_seed	`Optional[int]`	`123`
selection_strategy	`Optional[SelectionStrategy]`	`None`
split	`Optional[str]`	`None`
x_uids	`Optional[List[str]]`	`None`
filter_by_x_uids_not_in_batch	`Optional[bool]`	`False`
divide_x_uids_evenly_to_assignees	`Optional[bool]`	`False`

Returns

The list of created batches

Return type

List[Batch]

create_datasource

create_datasource(*data, uid_col=None, name=None, split='train', sync=True, run_checks=True*)

Creates a new data source withing the Dataset from either a filepath or a Pandas DataFrame.

If you provide a filepath: A file can be a CSV or Parquet file that either exists in the local filesystem, or is accessible via an S3-compatible API (such as MinIO, or AWS S3). Files must have a stable integer index column that is unique across all data sources in the dataset.

If you provide a DataFrame: The DataFrame must have a unique integer column that does not contain duplicates across other sources of data. All DataFrame column names must be strings.

The data must pass all validation checks to be registered as a valid data source. If a DataFrame fails to pass all data validation checks, the upload will fail and the data source will not be registered.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.create_datasource("my_data.csv", uid_col="id",
split="train")
+0.07s Starting data ingestion
+1.85s Ingesting data
+2.05s Data ingestion complete
1 # UID of the datasource

>>> my_dataset.create_datasource(df, uid_col="id", name="dataframe-
data", split="train")
+0.07s Starting data ingestion
+1.85s Ingesting data
+2.05s Data ingestion complete
1
```

Parameters

Name	Type	Default	Info
data	`Union[str, DataFrame]`		Either: - A path to a file in the local filesystem, or a path to an S3-compatible API, by default None. If filepath is not provided, a DataFrame must be provided instead - A Pandas DataFrame, by default None. If df is not provided, a filepath must be provided instead.
uid_col	`Optional[str]`	`None`	Name of the UID column for this data. The values in this column must be unique non-negative integers that are not duplicated across files. If not specified, the UID column will be generated in the server side.
name	`Optional[str]`	`None`	The name to give this data source. If not provided, the name of the file will be used, by default None. Adding a name is strongly recommended when uploading a DataFrame.
split	`str`	`'train'`	The name of the data split this data belongs to, by default Splits.train.
sync	`bool`	`True`	Whether execution should be blocked by this function, by default True. Note that Dataset().datasources may not

			be updated immediately if sync=False.
run_checks	`bool`	`True`	Whether we should run datasource checks. Recommended for safety, by default True.

Returns

If sync is True, returns the integer UID of the datasource. If sync is False, returns a job ID that can be monitored with `sf.poll_job_id`

Return type

`Union[str, int]`

create_label_schema

create_label_schema(*name, data_type, task_type, label_map, multi_label=False, description=None, label_column=None, label_descriptions=None, primary_field=None*)

Create a label schema associated with this dataset.

This is the recommended entrypoint for creating label schemas.

Parameters

Name	Type	Default	Info
name	`str`		The name of the label schema.
data_type	`str`		The data type of the label schema.
task_type	`str`		The task type of the label schema.

label_map	`Union[Dict[str, int], List[str]]`		A dictionary mapping label names to their integer values, or a list of label names.
multi_label	`bool`	`False`	Whether the label schema is a multi-label schema, by default False.
description	`Optional[str]`	`None`	A description of the label schema, by default None.
label_column	`Optional[str]`	`None`	The name of the column that contains the labels, by default None.
label_descriptions	`Optional[Dict[str, str]]`	`None`	A dictionary mapping label names to their descriptions, by default None.
primary_field	`Optional[str]`	`None`	The primary field of the label schema, by default None.

Returns

The label schema object

Return type

LabelSchema

delete

classmethod **delete**(*dataset, force=False*)

Delete a dataset based on the provided identifier

The operation will fail if any applications use this Dataset

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> Dataset.delete("my-dataset")
Successfully deleted dataset my-dataset with UID 0.
```

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		Name or UID of the dataset to delete.
force	`bool`	`False`	If True, delete any applications using the Dataset as well.

Return type

`None`

delete_datasource

delete_datasource(*datasource_uid, force=False, sync=True*)

Delete a data source. Calling delete_datasource will fully remove the data source from the dataset.

⚠ WARNING

The operation will not be permitted if any applications are using the data source to avoid breaking downstream applications. If you are sure you want to delete the data source, use the flag `force=True` to override this check. This function may take a while.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.delete_datasource(1)
Successfully deleted datasource with UID 1.
```

Parameters

Name	Type	Default	Info
datasource_uid	`int`		UID of the data source to delete. See all datasources for this dataset by viewing self.datasources.
force	`bool`	`False`	boolean allowing one to force deletion of a datasource even if that datasource has dependent assets (ground truth, annotations, etc), by default false.
sync	`bool`	`True`	Poll job status and block until complete, by default true.

Returns

Optionally returns job_id if sync mode is turned off

Return type

`Optional[str]`

get

classmethod **get**(*dataset*)

Fetches an already-created Dataset from Snorkel Flow and returns a Dataset object that can be used to interact with files and data

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
Successfully retrieved dataset my-dataset with UID 0 in workspace
0.
```

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		Either the name or UID of the dataset. A list of all accessible datasets can be retrieved with `Dataset.list()`

Returns

A Dataset object that can be used to interact with files and data in Snorkel Flow.

Return type

Dataset

get_dataframe

get_dataframe(*split=None, max_rows=10, target_columns=None, datasource_uid=None, use_source_index=True*)

Read the Dataset’s data into an in-memory Pandas DataFrame. If only a subset of columns are required, they can be specified with `target_columns`. Note that changes to the DataFrame will not be reflected in the Dataset. To change the actual data in the dataset, you must swap out the relevant data sources.

> NOTE
>
> By default, only 10 rows are read for memory safety. This limit can be increased by setting `max_rows` to a larger value, but this can be computationally intensive and may lead to unstable behavior.

NOTE

By default, we will return the original index column name the data source was uploaded with. However, certain SDK workflows might require an internal representation of the index column, such as the `snorkelai.sdk.develop.Deployment.execute` function. If you run into issues because of this, run `dataset.get_dataframe` with the `use_source_index` parameter set to `False`.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> df = my_dataset.get_dataframe(target_columns=["a", "b"])
<pd.DataFrame object with 10 rows and columns a, b>
>>> df = my_dataset.get_dataframe(datasource_uid=0, max_rows=None)
<pd.DataFrame object with 100 rows and columns a, b, c>
```

Parameters

Name	Type	Default	Info
split	`Optional[str]`	`None`	The data split to load, b default None (all splits) Other options are "trai "valid", and "test".
max_rows	`Optional[int]`	`10`	The maximum number rows to read, by defaul Use `max_rows=None` t fetch all rows. Warning setting this to a large va can be computationally intensive and may lead unstable behavior.
target_columns	`Optional[List[str]]`	`None`	A list of desired data columns, in case not al

			columns are required, default None.
datasource_uid	`Optional[int]`	`None`	Fetch a dataframe from particular `datasource_uid`. A li all datasource UIDs car retrieved with `Dataset().datasour` This can't be used with `split`.
use_source_index	`bool`	`True`	If true, returns the inde column that the data s was originally uploadec with. If false, returns th Snorkel Flow internal column name. True by default.

Returns

A Pandas DataFrame object displaying the data in this dataset

Return type

`pd.DataFrame`

list

static **list**()

Get a list of all Datasets. The returned list includes the Dataset UID, the Dataset name, and additional metadata used to keep track of the Dataset's properties.

Examples

```
>>> Dataset.list()
[
    {
        "name": "test-csv-str",
        "uid": 116,
        "datasources": []
    },
    ...
]
```

Returns

List of all dataset objects

Return type

List[Dataset]

update

update(*name=''*)

Update the metadata for this dataset. Only updating the name of this Dataset is currently supported. The new name for the dataset must be unique within the workspace.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get(dataset="my-dataset")
>>> my_dataset.update(name="my-new-dataset")
Successfully renamed dataset with UID 0 to my-new-dataset
```

Parameters

Name	Type	Default	Info
name	`str`	`''`	The new name for this dataset.

Returns

Confirmation string if this operation was successful

Return type

`str`

update_datasource_data

update_datasource_data(*old_datasource_uid, new_data, sync=True*)

This function allows you to replace the data of an existing data source with new data. This function can be used if you find an error in an existing value in a data source, or if you need to update values due to changes in your upstream data pipeline. This function requires that all row indexes in the new data source match the row indexes of the old data source. Additionally, all columns must have the same name and the same type.

If your goal is to change the number of columns, the number of rows, or the type of a column, you should consider using an Operator instead.

⚠ WARNING

This is a potentially dangerous operation, and may take a while to run. For safety, this will always run data source checks on the new data source. Applications and models that use the data source being replaced may become temporarily unavailable as computations are re-run over the new data, and might report different behavior. If you are unsure how to use this function, contact a Snorkel representative.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.datasources
[{"datasource_uid": 1, "datasource_name": "test.csv", "split":
"train"}]
>>> df = my_dataset.get_dataframe(datasource_uid=1, max_rows=None)
>>> df
|   |   a |   b |   c          |
|  0|   1 |   0 | bad_path.pdf|
>>> df.iloc[0, 2] = "good_path.pdf"
>>> my_dataset.update_datasource_data(1, df)
Successfully replaced data in datasource with UID 1.
```

Parameters

Name	Type	Default	
old_datasource_uid	`int`		The UID of the data source you w data sources for this dataset by v
new_data	`Union[str, DataFrame]`		Either (1) A path to a file in the loc compatible API, by default None. must be provided instead, or (2) A df is not provided, a filepath mus UIDs of the new data must exactl Use `dataset.get_dataframe(data` to see the existing data.
sync	`bool`	`True`	Poll job status and block until all j

Returns

Returns a Job ID that can be polled if sync is False. Otherwise returns None

Return type

`Optional[str]`

Raises

ValueError – If the data provided is neither a valid file path or a valid Pandas DataFrame

update_datasource_split

update_datasource_split(*datasource_uid, split*)

Change the split of a data source that has already been uploaded to the dataset. This will impact how the data source is used in all future applications.

> ⚠ WARNING
>
> This will only impact the Dataset’s data source, and not existing derived data sources. To change the split within applications that have already been created, find the node’s derived data source UID by clicking on “Develop” > “View Data Sources” in the Snorkel Flow UI and use the `sf.update_datasource` function.

Examples

```
>>> from snorkelai.sdk.develop import Dataset
>>> my_dataset = Dataset.get("my-dataset")
>>> my_dataset.datasources
[{"datasource_uid": 1, "datasource_name": "test.csv", "split":
"train"}]
>>> my_dataset.update_datasource_split(1, "train")
[123, 456, 789]
```

Parameters

Name	Type	Default	Info
datasource_uid	`int`		The integer UID corresponding to the data source you wish to update. You can see a list of all data sources for this dataset by viewing self.datasources.

split	`str`		The new split to assign to this data source. Must be one of “train”, “test”, or “valid”.

Returns

Returns a list of model nodes that have been impacted by changing the split.

Return type

`List[int]`

property **batches***: List[Batch]*

A list of batches belonging to this Dataset.

property **datasources***: List[Dict[str, Any]]*

A list of data sources and associated metadata belonging to this Dataset.

property **label_schemas***: List[LabelSchema]*

A list of label schemas belonging to this Dataset.

property **mta_enabled***: bool*

Whether or not multi-task annotation is enabled for this dataset.

property **name***: str*

The human-readable name of the dataset.

property **uid***: int*

The unique integer identifier for the dataset within Snorkel Flow.

snorkelai.sdk.develop.Evaluator

class snorkelai.sdk.develop.Evaluator(**args*, ***kwargs*)

Bases: `BaseModel`, `ABC`

Base class for all evaluators.

An evaluator assesses a datapoint containing an AI application's response against a specific criteria. Evaluators can be of two types:

- CodeEvaluator: Code-based (using custom Python functions)
- PromptEvaluator: Prompt-based (using LLM prompts)

The goal of an evaluator is to categorize the datapoint into one of the criteria's labels, ultimately assigning the integer associated with the label as that datapoint's score. An evaluator can also assign a rationale for its score, which is used to explain the score.

Read more in the Evaluation overview.

Using the `Evaluator` class requires the following import:

```
from snorkelai.sdk.develop import Evaluator
```

Parameters

Name	Type	Default	Info
benchmark_uid	`int`		The unique identifier of the benchmark that contains the criteria. The `benchmark_uid` is visible in the URL of the benchmark page in the Snorkel GUI. For example, `https://YOUR-SNORKEL-INSTANCE/benchmarks/100/` indicates a benchmark with `benchmark_uid` of `100`.
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
evaluator_uid	`int`		The unique identifier for this evaluator.

__init__

__init__(*args, **kwargs)

Methods

`__init__`(*args, **kwargs)	
`create`(criteria_uid, **kwargs)	Creates a new evaluator for a criteria.
`execute`(split[, num_rows])	Runs the evaluator against all datapoints in the specified dataset split.
`get`(evaluator_uid)	Retrieves the evaluator for a given uid.
`get_execution_result`(execution_uid)	Retrieves the evaluation results and scores for a specific execution.
`get_executions`()	Retrieves all executions for this evaluator.
`get_versions`()	Retrieves all version names for this evaluator.
`poll_execution_result`(execution_uid[, sync])	Polls the evaluation job status and retrieves partial results.
`update`([version_name])	Updates the evaluator with a new version.

Attributes

`benchmark_uid`	
`criteria_uid`	
`evaluator_uid`	

create

abstract classmethod **create**(*criteria_uid, **kwargs*)

Creates a new evaluator for a criteria.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
**kwargs	`Any`		Additional parameters specific to the evaluator type.

Return type

`Evaluator`

execute

abstract **execute**(*split, num_rows=None, **kwargs*)

Runs the evaluator against all datapoints in the specified dataset split.

Parameters

Name	Type	Default	Info
split	`str`		The dataset split you want to evaluate.
num_rows	`Optional[int]`	`None`	The number of rows to evaluate. If `None`, evaluates all rows.
**kwargs	`Any`		Additional parameters specific to the evaluator type.

Return type

`int`

get

classmethod **get**(*evaluator_uid*)

Retrieves the evaluator for a given uid.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier for the evaluator.

Returns

The requested evaluator object.

Return type

Evaluator

Example

```
evaluator = Evaluator.get(evaluator_uid=300)
```

get_execution_result

abstract **get_execution_result**(*execution_uid*)

Retrieves the evaluation results and scores for a specific execution.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The unique identifier of the execution you want to get results for.

Return type

`Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]

get_executions

abstract **get_executions**()

Retrieves all executions for this evaluator.

Return type

`List`[`Dict`[`str`, `Any`]]

get_versions

abstract **get_versions**()

Retrieves all version names for this evaluator.

Return type

`List`[`str`]

poll_execution_result

abstract **poll_execution_result**(*execution_uid, sync=False*)

Polls the evaluation job status and retrieves partial results.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The unique identifier of the execution you want to poll for.
sync	`bool`	`False`	Whether to wait for the job to complete.

Return type

`Tuple`[`str`, `Dict`[`str`, `Dict`[`str`, `Union`[`str`, `int`, `float`, `bool`]]]]

update

abstract **update**(*version_name=None, **kwargs*)

Updates the evaluator with a new version.

Parameters

Name	Type	Default	Info
version_name	`Optional[str]`	`None`	The name for the new version. If not provided, a default name will be generated.
**kwargs	`Any`		Additional parameters specific to the evaluator type.

Return type

`str`

benchmark_uid: *`int`*

criteria_uid: *`int`*

evaluator_uid: *`int`*

snorkelai.sdk.develop.JsonExportConfig

class snorkelai.sdk.develop.JsonExportConfig

Bases: `object`

__init__

__init__()

Methods

`__init__`()	

snorkelai.sdk.develop.LabelSchema

class snorkelai.sdk.develop.LabelSchema(*name, uid, dataset_uid, label_map, description, is_text_label=False*)

Bases: `object`

The LabelSchema object represents a label schema in Snorkel Flow. Currently, this interface only represents Dataset-level (not Node-level) label schemas.

__init__

__init__(*name, uid, dataset_uid, label_map, description, is_text_label=False*)

Create a label schema object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods

Parameters

Name	Type	Default	Info
name	`str`		The name of the label schema.
uid	`int`		The UID for the label schema within Snorkel Flow.
dataset_uid	`int`		The UID for the dataset within Snorkel Flow.
label_map	`Dict[str, int]`		The label map of the label schema.
description	`Optional[str]`		The description of the label schema.
is_text_label	`bool`	`False`	Whether the label schema is a text label schema.

Methods

`__init__`(name, uid, dataset_uid, label_map, ...)	Create a label schema object in-memory with necessary properties.
`copy`(name[, description, label_map, ...])	Copy a label schema.
`create`(dataset_uid, name, data_type, ...[, ...])	Create a label schema for a dataset.
`delete`(label_schema)	Delete a label schema by name or UID.
`get`(label_schema)	Retrieve a label schema by name or UID.

Attributes

`dataset_uid`	The UID for the dataset within Snorkel Flow.
`description`	The description of the label schema.
`is_text_label`	Whether the label schema is a text label schema.
`label_map`	The label map of the label schema.
`name`	The name of the label schema.
`uid`	The UID for the label schema within Snorkel Flow.

copy

copy(*name, description=None, label_map=None, label_descriptions=None, updated_label_schema=None*)

Copy a label schema.

Parameters

Name	Type	Default	Info
name	`str`		The name of the new label schema.

description	`Optional[str]`	`None`	The description of the new label schema.
label_map	`Optional[Dict[str, int]]`	`None`	The label map of the new label schema.
label_descriptions	`Optional[Dict[str, str]]`	`None`	The label descriptions of the new label schema.
updated_label_schema	`Optional[Dict[str, str]]`	`None`	The update mapping to apply to the new label schema. This is a dictionary mapping label names for the current label schema to those for the new label schema. If a label for the current label schema is removed, it is mapped to None. Examples: 1. Rename “old_1” to “new_1” and remove “old_2”: {“old_1”: “new_1”, “old_2”: None} 2. Merge “old_1” and “old_2” to “new_1”: {“old_1”: “new_1”, “old_2”:

			"new_1"} 3. Split "old_1" to "new_1" and "new_2", and keep assets labeled as "old_1" at "new_1": {"old_1": "new_1"} 4. Add "new_3": None (no change to the existing assets).

Returns

The new label schema object

Return type

LabelSchema

create

classmethod **create**(*dataset_uid, name, data_type, task_type, label_map, multi_label=False, description=None, label_column=None, label_descriptions=None, primary_field=None, is_text_label=False, allow_overlapping=None*)

Create a label schema for a dataset.

Typically, Dataset.create_label_schema() is the recommended entrypoint for creating label schemas.

Parameters

Name	Type	Default	Info
dataset_uid	`int`		The UID for the dataset within Snorkel Flow.

name	`str`		The name of the label schema.
data_type	`str`		The data type of the label schema.
task_type	`str`		The task type of the label schema.
label_map	`Union[Dict[str, int], List[str]]`		A dictionary mapping label names to their integer values, or a list of label names.
multi_label	`bool`	`False`	Whether the label schema is a multi-label schema, by default False.
description	`Optional[str]`	`None`	A description of the label schema, by default None.
label_column	`Optional[str]`	`None`	The name of the column that contains the labels, by default None.
label_descriptions	`Optional[Dict[str, str]]`	`None`	A dictionary mapping label names to their descriptions, by default None.
primary_field	`Optional[str]`	`None`	The primary field of the label schema, by default None.

is_text_label	`bool`	`False`	Whether the label schema is a text label schema, by default False.
allow_overlapping	`Optional[bool]`	`None`	Enable overlapping labels at the same token position. Defaults to None. Sequence tagging only.

Returns

The label schema object

Return type

LabelSchema

delete

classmethod **delete**(*label_schema*)

Delete a label schema by name or UID.

Parameters

Name	Type	Default	Info
label_schema	`Union[str, int]`		The name or UID of the label schema.

Return type

`None`

get

classmethod **get**(*label_schema*)

Retrieve a label schema by name or UID.

Parameters

Name	Type	Default	Info
label_schema	`Union[str, int]`		The name or UID of the label schema.

Returns

The label schema object

Return type

LabelSchema

Raises

ValueError – If no label schema is found with the given name or UID

property **dataset_uid***: int*

The UID for the dataset within Snorkel Flow.

property **description***: str | None*

The description of the label schema.

property **is_text_label***: bool*

Whether the label schema is a text label schema.

property **label_map***: Dict[str, int]*

The label map of the label schema.

property **name***: str*

The name of the label schema.

property **uid***: int*

The UID for the label schema within Snorkel Flow.

snorkelai.sdk.develop.Node

class snorkelai.sdk.develop.Node(*uid*, *application_uid*, *config*)

Bases: `ABC`

The Node object represents atomic data processing units in Snorkel Flow.

Nodes Quickstart

```
from snorkelai.sdk.develop import Node
# Get a Node object by its UID
my_node = Node.get(123)

# Get a dataframe
df = my_node.get_dataframe()
```

Nodes Concepts

Nodes

A Node is a unit of data processing in Snorkel Flow. Nodes are split into two broad categories, *Operators* and *Models*. Operator nodes apply transformations to the data, such as adding a column, modifying a column's values, combining multiple dataframes, or filtering rows. Model nodes are the machine learning hubs in the data pipeline, where you can query and finetune foundation models, explore your data, and train your own models. Nodes are stitched together in a particular order, which dictates how your data flows from your source Dataset all the way to your final desired outputs. The Application DAG (directed acyclic graph) helps visualize the order that the nodes are organized in.

Fetching Data in the Notebook

The Node object is the primary way to fetch data in the Snorkel Flow Notebook. The Node object has a `get_dataframe()` method, which returns a Pandas DataFrame corresponding to what the DAG sees when it is processing data at that point in the graph. This is useful for debugging and understanding how your data is being transformed throughout the data development process. However, since the Notebook environment often has fewer compute resources than the core Snorkel Flow backend, there are some caveats to be aware of when interacting with your data this way. By default, the `get_dataframe()` method will return a maximum of 10 rows of data, to prevent the

Notebook from running out of memory. This safety latch can be manually overridden, but should be done with caution.

__init__

__init__(*uid, application_uid, config*)

Methods

`__init__`(uid, application_uid, config)	
`get`(node_uid)	Fetches a node by its UID.
`get_dataframe`()	Retrieve the data being passed directly through this node.

Attributes

`application_uid`	The unique identifier for the application this node belongs to
`config`	Returns the detailed configuration information for this node
`uid`	The unique identifier for this node

get

classmethod **get**(*node_uid*)

Fetches a node by its UID. Returns either a ModelNode or OperatorNode object, which can be used to fetch and manipulate node-level data.

Parameters

Name	Type	Default	Info
node_uid	`int`		The UID for the particular node. You can find this UID by clicking on the node in the DAG view in Developer Studio, or by looking at the `node_dag` field of the return value of `snorkelai.sdk.client.get_application()`.

Returns

A `Node` object, an instance of `OperatorNode`.

Return type

Node

get_dataframe

abstract **get_dataframe**()

Retrieve the data being passed directly through this node.

This dataframe is not the same as the dataframe returned by `Dataset.get_dataframe()`. While `Dataset.get_dataframe()` returns the source data, the dataframe returned by `Node.get_dataframe()` has also undergone all the preprocessing/DAG transformations up to this point in the processing pipeline.

Returns

A DataFrame of the data being passed directly through this node. This DataFrame is the result of all preprocessing in the DAG pipeline up to this point.

Return type

`pd.DataFrame`

property **application_uid***: int*

The unique identifier for the application this node belongs to

property **config***: Dict[str, Any]*

Returns the detailed configuration information for this node

property **uid***: int*

The unique identifier for this node

snorkelai.sdk.develop.OperatorNode

class snorkelai.sdk.develop.OperatorNode(*uid, application_uid, config*)

Bases: `Node`

OperatorNode class represents a non-model, operator node.

__init__

__init__(*uid, application_uid, config*)

Methods

`__init__`(uid, application_uid, config)	
`get`(node_uid)	Fetches a node by its UID.
`get_dataframe`([max_input_rows, ...])	Retrieve the data being passed directly through this node.

Attributes

`application_uid`	The unique identifier for the application this node belongs to
`config`	Returns the detailed configuration information for this node
`uid`	The unique identifier for this node

get_dataframe

get_dataframe(*max_input_rows=10, datasource_uids=None, partition=None*)

Retrieve the data being passed directly through this node. By default, this function will only process a maximum of 10 rows of data, to prevent the Notebook from running out of memory. To override this limit, set `max_input_rows` to a higher value.

This dataframe is not the same as the dataframe returned by `Dataset.get_dataframe()`. While `Dataset.get_dataframe()` returns the source data, the dataframe returned by `Node.get_dataframe()` has also undergone all the preprocessing/DAG transformations up to this point in the processing pipeline.

Parameters

Name	Type	Default	Info
max_input_rows	`int`	`10`	The number of rows that should be pushed through this node, by default 10.
datasource_uids	`Optional[List[int]]`	`None`	A list of datasource UIDs to process, useful if you have some specific datasources you want to examine, by default None. See the `Dataset` class for more information on fetching a datasource UID.
partition	`Optional[int]`	`None`	A specific file partition to process, by default None. Only applicable if the source dataset files are in a readily partitioned format.

Returns

A DataFrame displaying the results when the source dataset is pushed through this node.

Return type

`pd.DataFrame`

snorkelai.sdk.develop.PromptEvaluator

class snorkelai.sdk.develop.PromptEvaluator(**args, **kwargs*)

Bases: `Evaluator`

An evaluator that uses LLM prompts to assess model outputs.

This evaluator type is known as an LLM-as-a-judge (LLMAJ). A prompt evaluator uses LLM prompts to evaluate datapoints containing AI application responses, categorizing them into one of a criteria's labels by assigning the corresponding integer score and optional rationale.

Prompt evaluator execution via the SDK is not yet supported. Please use the GUI to run prompt evaluators.

Read more about LLM-as-a-judge prompts.

Using the `PromptEvaluator` class requires the following import:

```
from snorkelai.sdk.develop import PromptEvaluator
```

__init__

__init__(**args, **kwargs*)

Methods

`__init__`(*args, **kwargs)	
`create`(criteria_uid, **kwargs)	Creates a new prompt evaluator for a criteria.
`execute`(split[, num_rows, version_name, sync])	Executes the prompt evaluator against a dataset split.
`get`(evaluator_uid)	Retrieves a prompt evaluator for a given uid.
`get_execution_result`(execution_uid)	Retrieves the evaluation results for a specific evaluation execution.
`get_executions`()	Retrieves all executions for this prompt evaluator.
`get_versions`()	Gets all version names for a prompt evaluator.

`poll_execution_result`(execution_uid[, sync])	Polls the job status and retrieves partial results.
`update`([version_name])	Creates a new prompt version for a criteria and updates the evaluator to point to the new prompt version.

Attributes

`benchmark_uid`	
`criteria_uid`	
`evaluator_uid`	
`prompt_workflow_uid`	

create

classmethod **create**(*criteria_uid, **kwargs*)

Creates a new prompt evaluator for a criteria.

Parameters

Name	Type	Default	Info
criteria_uid	`int`		The unique identifier of the criteria that this evaluator assesses.
**kwargs	`Any`		user_prompt: Optional[str] = None The user prompt to use for the evaluator. At least one of user_prompt or system_prompt must be provided. system_prompt: Optional[str] = None The system prompt to use for the evaluator. At least one of user_prompt or system_prompt must be provided. model_name: str The model to use for the evaluator.

fm_hyperparameters: Optional[Dict[str, Any]] = None

The hyperparameters to use for the evaluator. These are provided directly to the model provider.
For example, OpenAI supports the *response_format* hyperparameter. It can be provided in the following way:

```
PromptEvaluator.create(
    criteria_uid=100,
    user_prompt="User prompt",
    system_prompt="System prompt",
    model_name="gpt-4o-mini",
    fm_hyperparameters={
        "response_format": {
            "type": "json_object",
        }
    }
)
```

Or a more sophisticated example:

```
PromptEvaluator.create(
    criteria_uid=100,
    user_prompt="User
prompt",
    system_prompt="System
prompt",
    model_name="gpt-4o-mini",
    fm_hyperparameters={
        "response_format": {
            "type":
"json_schema",
            "json_schema": {
                "name":
"math_reasoning",
                "schema": {
                    "type":
"object",

"properties": {

"steps": {

"type": "array",

"items": {

"type": "object",

"properties": {

"explanation": {"type":
"string"},

"output": {"type": "string"}

},

"required": ["explanation",
"output"],

"additionalProperties": False
                            }
                        },
"final_answer": {"type":
"string"}
                    },

"required": ["steps",
"final_answer"],

"additionalProperties": False
                },
                "strict":
True
            }
        }
    }
)
```

Returns

A PromptEvaluator object representing the new evaluator.

Return type

PromptEvaluator

Raises

ValueError – If one of user_prompt, system_prompt is not provided. If model_name is not provided.

execute

execute(*split, num_rows=None, version_name=None, sync=False, **kwargs*)

Executes the prompt evaluator against a dataset split.

This method runs the prompt against the specified dataset split. If no version name is specified, it uses the latest version.

Parameters

Name	Type	Default	Info

split	`str`		The dataset split to evaluate on.
num_rows	`Optional[int]`	`None`	The number of rows to evaluate on.
version_name	`Optional[str]`	`None`	The version name to use for the execution. If not provided, it uses the latest version.
sync	`bool`	`False`	Whether to wait for the execution to complete.

Returns

The execution uid.

Return type

`int`

Example

Example 1

Execute the latest prompt version and poll for results:

```
import time

evaluator = PromptEvaluator.get(evaluator_uid=300)
prompt_execution_uid = evaluator.execute(split="test",
num_rows=100)
while True:
    status, results =
evaluator.poll_execution_result(prompt_execution_uid, sync=False)
    print(f"Job status: {status}")
    if status == "completed" or status == "failed":
        break
    if results:
        print(f"Partial results: {results}")
    time.sleep(10)

print(f"Final results: {results}")
```

Example 2

Execute a specific prompt version and wait for results:

```
evaluator = PromptEvaluator.get(evaluator_uid=300)
prompt_execution_uid = evaluator.execute(split="train",
num_rows=20, version_name="v1.0")
status, results =
evaluator.poll_execution_result(prompt_execution_uid, sync=True)
print(f"Status: {status}")
print(f"Results: {results}")
```

get

classmethod **get**(*evaluator_uid*)

Retrieves a prompt evaluator for a given uid.

Parameters

Name	Type	Default	Info
evaluator_uid	`int`		The unique identifier for the evaluator.

Returns

A PromptEvaluator instance.

Return type

PromptEvaluator

Raises

ValueError – If the evaluator with the given uid is not a PromptEvaluator.

get_execution_result

get_execution_result(*execution_uid*)

Retrieves the evaluation results for a specific evaluation execution.

This method reads the evaluation results for the given evaluation execution UID. If the execution is in progress, it will return partial results.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The evaluation execution UID to get results for.

Returns

A dictionary mapping x_uids to their evaluation results. The evaluation results for each x_uid are a dictionary with the following keys: - “score”: The score for the datapoint - “rationale”: The rationale for the score

Return type

`Dict[str, Dict[str, EvaluationScoreType]]`

get_executions

get_executions()

Retrieves all executions for this prompt evaluator.

This method fetches all executions that have been run using this evaluator. Executions are returned in chronological order, with the oldest execution first.

The dictionary contains the following keys: :rtype: `List`[`Dict`[`str`, `Any`]]

- `execution_uid`: The execution UID
- `created_at`: The timestamp when the execution was created
- `prompt_version_name`: The name of the prompt version used for the execution

Example

Example 1

Get all executions for an evaluator:

```
evaluator = PromptEvaluator.get(evaluator_uid=300)
executions = evaluator.get_executions()
for execution in executions:
    print(f"Execution {execution['execution_uid']}:
{execution['created_at']}")
```

get_versions

get_versions()

Gets all version names for a prompt evaluator.

Returns

A list of version names for the prompt evaluator.

Return type

`List[str]`

poll_execution_result

poll_execution_result(*execution_uid, sync=False*)

Polls the job status and retrieves partial results.

This method checks the current status of the evaluation job and returns both the job status and any available results. The current status can be `running`, `completed`, `failed`, `cancelled`, or `unknown`.

Parameters

Name	Type	Default	Info
execution_uid	`int`		The prompt execution UID to poll for.
sync	`bool`	`False`	Whether to wait for the job to complete. If `False`, returns immediately with current status and partial results.

Return type

`Tuple[str, Dict[str, Dict[str, Union[str, int, float, bool]]]]`

Example

Example 1

Poll for job status and partial results:

```
evaluator = PromptEvaluator.get(evaluator_uid=300)
prompt_execution_uid = evaluator.execute(split="test",
num_rows=100)
while True:
    status, results =
evaluator.poll_execution_result(prompt_execution_uid, sync=False)
    print(f"Job status: {status}")
    if results:
        print(f"Partial results: {results}")
    if status == "completed" or status == "failed":
        break
print(f"Final results: {results}")
```

update

update(*version_name=None, **kwargs*)

Creates a new prompt version for a criteria and updates the evaluator to point to the new prompt version.

Parameters

Name	Type	Default	Info
version_name	`Optional[str]`	`None`	The name for the new prompt vers default name will be generated.
**kwargs	`Any`		user_prompt: Optional[str] = None The user prompt to use for t one of user_prompt or syste provided. system_prompt: Optional[str] = Nc

The system prompt to use fo
one of user_prompt or syste
provided.

model_name: str

The model to use for the eva

fm_hyperparameters: Optional[Dic

The hyperparameters to use
are provided directly to the r
For example, OpenAI suppor
hyperparameter. It can be pr
way:

```
evaluator =
PromptEvaluator.get(
evaluator.update(
    version_name="Nev
    user_prompt="Use
    system_prompt="Sy
    model_name="gpt-
    fm_hyperparamete
        "response_fo
            "type": '
        }
    }
)
```

Or a more sophisticated exa

```
evaluator =
PromptEvaluator.get(
evaluator.update(
    version_name="Ne
    user_prompt="Use
    system_prompt="S
    model_name="gpt-
    fm_hyperparamete
        "response_fo
            "type": 
            "json_sc
                "nam
"math_reasoning",
                "sch



"array",


"type": "object",

"properties": {

"explanation": {"typ

"output": {"type": "


"required": ["explan
"output"],

"additionalPropertie



"final_answer": {"ty


["steps", "final_ans

"additionalPropertie
                },
```

				`"str`
				`}`
				`}`
				`}`
				`)`

Returns

The version name of the new prompt version.

Return type

`str`

Raises

ValueError – If one of user_prompt, system_prompt is not provided. If model_name is not provided.

benchmark_uid: *int*

criteria_uid: *int*

evaluator_uid: *int*

prompt_workflow_uid: *int*

snorkelai.sdk.develop.Slice

class snorkelai.sdk.develop.Slice(*dataset, slice_uid, name, description=None, config=None*)

Bases: `object`

Represents a slice within a Snorkel dataset for identifying and managing subsets of datapoints.

A slice is a logical subset of datapoints within a dataset that can be created either manually by adding specific datapoints, or programmatically using slicing functions defined through templates and configurations. Slices are essential for data analysis, model evaluation, and targeted data operations within Snorkel workflows.

Key capabilities:

- Manual datapoint management through add/remove operations
- Programmatic datapoint identification using configurable slicing functions

This class provides methods for creating, retrieving, updating, and managing slice membership. Slice objects should not be instantiated directly - use the `create()` or `get()` class methods instead.

For more information on slices and data management, see the Data Management Guide.

Parameters

Name	Type	Default	Info
dataset	`IdType`		The UID or name for the dataset within Snorkel.
slice_uid	`int`		The unique identifier for the slice within Snorkel.
name	`str`		The display name of the slice.
description	`str, optional`		A description of the slice’s purpose and contents.

config	`SliceConfig, optional`		Configuration defining slicing functions for programmatic datapoint identification.

__init__

__init__(*dataset, slice_uid, name, description=None, config=None*)

Create a Slice object in-memory with necessary properties. This constructor should not be called directly, and should instead be accessed through the `create()` and `get()` methods

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel.
slice_uid	`int`		The UID for the slice within Snorkel.
name	`str`		The name of the slice.
description	`Optional[str]`	`None`	The description of the slice.

Methods

`__init__`(dataset, slice_uid, name[, ...])	Create a Slice object in-memory with necessary properties.
`add_x_uids`(x_uids)	Add datapoints to a slice.
`create`(dataset, name[, description, config])	Create a slice for a dataset.
`get`(dataset, slice)	Retrieve a slice by UID.
`get_x_uids`()	Retrieve the UIDs of the datapoints in the slice.
`list`(dataset)	Retrieve all slices for a dataset.

`remove_x_uids`(x_uids)	Remove datapoints from a slice.
`update`([name, description, config])	Update the slice properties.

Attributes

`dataset_uid`	Return the UID of the dataset that the slice belongs to
`description`	Return the description of the slice
`name`	Return the name of the slice
`slice_uid`	Return the UID of the slice

add_x_uids

add_x_uids(*x_uids*)

Add datapoints to a slice.

Parameters

Name	Type	Default	Info
x_uids	`List[str]`		List of UIDs of the datapoints you want to add to the slice.

Raises

Exception – If other server errors occur during the operation

Return type

`None`

create

classmethod **create**(*dataset, name, description='', config=None*)

Create a slice for a dataset. Slices are use to identify a subset of datapoints in a dataset. You can add datapoints to a slice manually, or if you define a config, you can add datapoints programmatically. Slice membership can contain both manual and programmatic identified datapoints.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel Flow.
name	`str`		The name of the slice.
description	`str`	`''`	A description of the slice, by default the empty string.
config	`Optional[SliceConfig]`	`None`	A SliceConfig object, by default None, you can reference the schema in the *template* module for constructing this config. This config is used to define the Slicing Function (templates and graph) for the slice, allowing it to programmatically add datapoints to the slice membership.

Returns

The slice object

Return type

Slice

Raises

- **ValueError** – If the dataset doesn't exist or cannot be found by name/UID
- **ValueError** – If the slice name is a reserved name or already exists for the dataset
- **ValueError** – If there are other validation or server errors during slice creation

Examples

```
>>> from snorkelai.templates.keyword_template import
KeywordTemplateSchema
>>> from snorkelai.sdk.develop.slices import Slice, SliceConfig
>>> from snorkelai.sdk.utils.graph import DEFAULT_GRAPH
>>> Slice.create(
>>>     dataset=dataset_uid,
>>>     name="slice_name",
>>>     description="description",
>>>     config=SliceConfig(
>>>         templates=[
>>>         {
>>>             "transform_type": "dataset_template_filter",
>>>             "config": {
>>>                 "transform_config_type": "filter_schema",
>>>                 "filter_type": "text_template",
>>>                 "filter_config_type": "dataset_text_template",
>>>                 "dataset_uid": 1,
>>>                 "template_config": {
>>>                     "template_type": "keyword",
>>>                     "field": "text_col",
>>>                     "keywords": ["keyword1", "keyword2"],
>>>                     "operator": "CONTAINS",
>>>                     "case_sensitive": False,
>>>                     "tokenize": True,
>>>                 },
>>>              },
>>>         },
>>>         ],
>>>         graph=DEFAULT_GRAPH,
>>>     ),
>>> )
```

get

classmethod **get**(*dataset, slice*)

Retrieve a slice by UID.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel Flow.
slice	`Union[str, int]`		The UID or name of the slice.

Returns

The slice object

Return type

Slice

Raises

ValueError – If no slice is found with the given UID

get_x_uids

get_x_uids()

Retrieve the UIDs of the datapoints in the slice.

Returns

List of UIDs of the datapoints in the slice

Return type

`List[str]`

Raises

Exception – If other server errors occur during the operation

list

classmethod **list**(*dataset*)

Retrieve all slices for a dataset.

Parameters

Name	Type	Default	Info
dataset	`Union[str, int]`		The UID or name for the dataset within Snorkel Flow.

Returns

A list of all the slices available for that dataset

Return type

List[Slice]

Raises

ValueError – If no dataset is found with the given id

remove_x_uids

remove_x_uids(*x_uids*)

Remove datapoints from a slice.

Parameters

Name	Type	Default	Info
x_uids	`List[str]`		List of UIDs of the datapoints you want to remove from the slice.

Raises

Exception – If other server errors occur during the operation

Return type

`None`

update

update(*name=None, description=None, config=None*)

Update the slice properties.

Parameters

Name	Type	Default	Info
name	`Optional[str]`	`None`	The new name for the slice, by default None.
description	`Optional[str]`	`None`	The new description for the slice, by default None.
config	`Optional[SliceConfig]`	`None`	A SliceConfig object with the new configuration for the slice, by default None.

Raises

ValueError – If there are other errors during slice update

Return type

`None`

property **dataset_uid***: int*

Return the UID of the dataset that the slice belongs to

property **description***: str | None*

Return the description of the slice

property **name***: str*

Return the name of the slice

property **slice_uid***: int*

Return the UID of the slice

snorkelai.sdk.utils

Other general utilities.

Functions

`open_file`(path[, mode])	Opens a file at the specified path and returns the corresponding file object for user files.
`resolve_data_path`(path)	Resolve a MinIO path to a local file path.

snorkelai.sdk.utils.open_file

WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.utils.open_file(*path, mode='r', **kwargs*)

Opens a file at the specified path and returns the corresponding file object for user files.

Currently supports MinIO URLs in the following format: `minio://{bucket}/{key}`, local file paths, and S3 URLs.

New MinIO paths paths are resolved to the current workspace by default, ensuring seamless compatibility. Workspace-scoped paths (e.g., *minio://workspace-1/file.txt*) are also supported if explicitly provided and match the current workspace.

Existing non-workspace-scoped MinIO paths paths will continue to work as expected. However, if a file with the same name is created in a workspace-scoped path, it will take priority moving forward.

Examples

```
>>> # Open a file in MinIO
>>> f = open_file("minio://my-bucket/my-key")
```

```
>>> # Open a file in MinIO outside of the in-platform Notebook
>>> f = open_file(
        "minio://my-bucket/my-key",
        key={MINIO-ACCESS-KEY},
        secret={MINIO-SECRET-KEY},
        client_kwargs={"endpoint_url": {MINIO-URL}}
    )
```

```
>>> # Open a local file
>>> f = open_file("./path/to/file")
```

```
>>> # Open a file in S3
>>> f = open_file("s3://my-bucket/my-key", key={YOUR-S3-ACCESS-KEY},
secret={YOUR-S3-ACCESS-KEY})
```

Parameters

Name	Type	Default	Info
path	`str`		The path of the file to be opened.
mode	`str`	`'r'`	`'w'`, `'rb'`, etc.
kwargs	`Optional[Dict[str, Any]]`		Extra options that make sense to a particular storage connection, e.g. host, port, username, password, etc. These options are passed directly to `fsspec.open`.

Returns

A file object opened in the specified mode

Return type

`OpenFile`

snorkelai.sdk.utils.resolve_data_path

⚠ WARNING

This SDK function is not supported in Snorkel AI Data Development Platform v25.5 and later versions and will be removed in a future release. For assistance finding alternative approaches, please contact your Snorkel support team.

snorkelai.sdk.utils.resolve_data_path(*path*)

Resolve a MinIO path to a local file path.

This method can resolve a MinIO path only if it is called from the in-app notebook.

Parameters

Name	Type	Default	Info
path	`str`		MinIO path (e.g., “minio://path/to/file.parquet”).

Returns

File path

Return type

`str`

Examples

```
>>> import pandas as pd
>>> from snorkelai.sdk.utils import resolve_data_path
>>> minio_path = "minio://path/to/file.parquet"
>>> resolved_path = resolve_data_path(minio_path)
>>> df = pd.read_parquet(resolved_path)
```