Meerkat

Release 0.2.5

The Meerkat Team

Oct 07, 2022

CONTENTS

Meerkat provides fast and flexible data structures for working with complex machine learning datasets. It is designed to house your data throughout the machine learning lifecycle – along the way enabling interactive data exploration, cross-modal training, and fine-grained error analysis.

CHAPTER

ONE

INSTALLATION

CHAPTER

TWO

NEXT STEPS

Get started with Meerkat by following along on Google Colab.

Walkthrough Notebook

Learn more about the motivation behind Meerkat and what it enables.

Introductory Blog Post

2.1 User Guide

2.1.1 Meerkat Basics

Introduction to Data Structures

Meerkat provides two data structures, the column and the datapanel, that together help you build, manage, and explore machine learning datasets . Everything you do with Meerkat will involve one or both of these data structures, so we begin this user guide with their high-level introduction.

Column

A column is a sequential data structure (analagous to a Series in Pandas or a Vector in R). Meerkat supports a diverse set of column types (*e.g.* *NumpyArrayColumn*, *ImageColumn*), each intended for different kinds of data. To see a list of the core column types and their capabilities, see *Overview of Column Types*.

Below we create a simple column to hold a set of images stored on disk. To create it, we simply pass filepaths to the *ImageColumn* constructor.

```
In [1]: import meerkat as mk

In [2]: img_col = mk.ImageColumn(
   ...:     ["img_0.jpg", "img_1.jpg", "img_2.jpg"],
   ...:     base_dir=abs_path_to_img_dir
   ...: )
   ...:

In [3]: img_col
Out[3]: ImageColumn(LambdaOp(args...rn_index=None))
```

All Meerkat columns are subclasses of *AbstractColumn* and share a common interface, which includes `__len__()`, `__getitem__()`, `__setitem__()`, *filter()*, `map()`, and *concat()*. Below we get the length of the column we just created.

```
In [4]: len(img_col)
Out[4]: 3
```

Certain column types may expose additional functionality. For example, *NumpyArrayColumn* inherits most of the functionality of an ndarray.

```
In [5]: id_col = mk.NumpyArrayColumn([0, 1, 2])

In [6]: id_col.sum()
Out[6]: 3

In [7]: id_col == 1
Out[7]: NumpyArrayColumn(array([False,  True, False]))
```

To see the full list of methods available to a column type,

If you don't know which column type to use, you can just pass a familiar data structure like a `list`, `np.ndarray`, `pd.Series`, and `torch.Tensor` to *from_data()* and Meerkat will automatically pick an appropriate column type.

```
In [8]: import torch

In [9]: tensor = torch.tensor([1,2,3])

In [10]: mk.AbstractColumn.from_data(tensor)
Out[10]: TensorColumn(tensor([1, 2, 3]))
```

DataPanel

A `DataPanel` is a collection of equal-length columns (analagous to a DataFrame in Pandas or R). DataPanels in Meerkat are used to manage datasets and per-example artifacts (*e.g.* model predictions and embeddings).

Below we combine the columns we created above into a single DataPanel. We also add an additional column containing labels for the images. Note that we can pass non-Meerkat data structures like `list`, `np.ndarray`, `pd.Series`, and `torch.Tensor` directly to the DataPanel constructor and Meerkat will infer the column type. We do not need to first convert to a Meerkat column.

```
In [11]: dp = mk.DataPanel(
   ....:     {
   ....:         "img": img_col,
   ....:         "label": ["boombox", "truck", "dog"],
   ....:         "id": id_col,
   ....:     }
   ....: )
   ....:

In [12]: dp
Out[12]: DataPanel(nrows: 3, ncols: 3)
```

Read on to learn how we access the data in Columns and DataPanels.

Data Selection

As discussed in the *Introduction to Data Structures*, there are two key data structures in Meerkat: the Column and the DataPanel. In this guide, we'll demonstrate how to access the data stored within them.

Throughout, we'll be selecting data from the following DataPanel, which holds the Imagenette dataset, a small subset of the original ImageNet. This DataPanel includes a column holding images, a column holding their labels, and a few others.

```
In [1]: import meerkat as mk

In [2]: dp = mk.datasets.get("imagenette")

In [3]: dp
Out[3]: DataPanel(nrows: 13394, ncols: 13)
```

Selecting Columns

The columns in a DataPanel are uniquely identified by `str` names. The code below displays the column names in the Imagenette datapanel we loaded above:

```
In [4]: dp.columns
Out[4]:
['path',
 'noisy_labels_0',
 'noisy_labels_1',
 'noisy_labels_5',
 'noisy_labels_25',
 'noisy_labels_50',
 'is_valid',
 'label_id',
 'label',
 'label_idx',
 'split',
 'img_path',
 'img']
```

Using these column names, we can pull out an individual column or a subset of them as a new DataPanel.

Selecting a Single Column: `str` -> *`AbstractColumn`*

To select a single column, we simply pass it's name to the index operator. For example,

```
In [5]: col = dp["label"]

In [6]: col
Out[6]: PandasSeriesColumn(0        cass... dtype: object)
```

Passing a `str` that isn't among the column names will raise a `KeyError`.

It may be helpful to think of a DataPanel as a dictionary mapping column names to columns. Indeed, a DataPanel implements other parts of the `dict` interface including *`keys()`*, *`values()`*, and *`items()`*. Unlike a dictionary, multiple columns in a DataPanel can be selected at once.

Selecting Multiple Columns: `Sequence[str]` -> *`DataPanel`*

You can select multiple columns by passing a list or tuple of column names. Doing so will return a new DataPanel with a subset of the columns in the original. For example,

```
In [7]: new_dp = dp[["label", "img"]]

In [8]: new_dp.columns
Out[8]: ['label', 'img']
```

Passing a `str` that isn't among the column names will raise a `KeyError`.

Copy vs. Reference

See *Copy vs. View Behavior* for more information.

You may be wondering whether the columns returned by indexing are copies of the columns in the original DataPanel. The columns returned by the index operator reference the same columns in the original DataPanel. This means that modifying the columns returned by the index operator will modify the columns in the original DataPanel.

Selecting Rows

In Meerkat, the rows of a DataPanel or Column are ordered. This means that rows are uniquely identified by their position in the DataPanel or Column (similar to how the elements of a Python List are uniquely identified by their position in the list).

Row indices range from 0 to the number of rows in the DataPanel or Column minus one. To see how many rows a DataPanel or a column has we can use `len()`. For example,

```
In [9]: len(dp)
Out[9]: 13394
```

Above we mentioned how a DataPanel could be viewed as a dictionary mapping column names to columns. Equivalently, it also may be helpful to think of a DataPanel as a list of dictionaries mapping column names to values. The DataPanel interface supports both of these views – under the hood, storage is organized so as to make both column and row accesses fast.

Selecting a Single Row from a DataPanel: `int -> Dict[str, Any]`

To select a single row from a DataPanel, we simply pass it's position to the index operator. For example,

```
In [10]: row = dp[2]

In [11]: row
Out[11]:
{'path': 'train/n02979186/n02979186_9715.JPEG',
 'noisy_labels_0': 'n02979186',
 'noisy_labels_1': 'n02979186',
 'noisy_labels_5': 'n02979186',
 'noisy_labels_25': 'n03417042',
 'noisy_labels_50': 'n03000684',
 'is_valid': False,
 'label_id': 'n02979186',
 'label': 'cassette player',
 'label_idx': 482,
 'split': 'train',
```

(continues on next page)

(continued from previous page)

```
 'img_path': 'train/n02979186/n02979186_9715.JPEG',
 'img': <PIL.Image.Image image mode=RGB size=96x130>}
```

Passing an `int` that is less than `0` or greater than `len(dp)` will raise an `IndexError`.

Notice how `row` contains a full PIL Image. With thousands of images in the dataset, it wouldn't make sense to hold all the images in memory. Instead, images are only loaded into memory at the moment they are selected.

Lazy Selection

What if we want to select a row without loading the image into memory? Meerkat supports lazy selection through the `lz` indexer.

```
In [12]: row = dp.lz[2]

In [13]: row
Out[13]:
{'path': 'train/n02979186/n02979186_9715.JPEG',
 'noisy_labels_0': 'n02979186',
 'noisy_labels_1': 'n02979186',
 'noisy_labels_5': 'n02979186',
 'noisy_labels_25': 'n03417042',
 'noisy_labels_50': 'n03000684',
 'is_valid': False,
 'label_id': 'n02979186',
 'label': 'cassette player',
 'label_idx': 482,
 'split': 'train',
 'img_path': 'train/n02979186/n02979186_9715.JPEG',
 'img': FileCell(fn=<meerkat.columns.file_column.FileLoader object at 0x7fc2979e6700>)}
```

Notice that instead of holding the image in memory, `row` holds a *`FileCell`* object. This object knows how to load the image into memory, but stops just short of doing so. Later on, when we want to access the image, we can use the :meth:``~meerkat.FileCell.get` method on the cell. For example,

```
In [14]: row["img"].get()
Out[14]: <PIL.Image.Image image mode=RGB size=96x130>
```

Lazy selection is critical for manipulating and managing DataPanels in Meerkat. It is discussed in more detail in the guide on *Lambda Columns and Lazy Selection*.

The same position-based indexing works for selecting a single cell from a Column.

Selecting a Single Cell from a Column: `int -> Any`

To select a single cell from a column, we pass it's position to the index operator. For example,

```
In [15]: col = dp["label"]

In [16]: col[2]
Out[16]: 'cassette player'
```

Passing an `int` that is less than `0` or greater than `len(dp["label"])` will raise an `IndexError`.

There are three different ways to select a subset of rows from a DataPanel: via `slice`, `Sequence[int]`, or `Sequence[bool]`.

Selecting Multiple Rows from a DataPanel: `slice` -> *DataPanel*

To select a set of contiguous rows from a DataPanel, we can use an integer slice `[start:end]`. The subset of rows will be returned as a new DataPanel.

```
In [17]: new_dp = dp[50:100]

In [18]: new_dp
Out[18]: DataPanel(nrows: 50, ncols: 13)
```

We can also use integer slices to select a set of evenly spaced rows from a DataPanel `[start:end:step]`. For example, below we select everyt tenth row from the first 100 rows in the DataPanel.

```
In [19]: new_dp = dp[0:100:10]

In [20]: new_dp
Out[20]: DataPanel(nrows: 10, ncols: 13)
```

Selecting Multiple Rows from a DataPanel: `Sequence[int]` -> *DataPanel*

To select multiple rows from a DataPanel we can also pass a list of `int`.

```
In [21]: small_dp = dp[[0, 2, 5, 8, 17]]

In [22]: small_dp
Out[22]: DataPanel(nrows: 5, ncols: 13)
```

Other valid sequences of `int` that can be used to index are:

- `Tuple[int]` – a tuple of integers.
- `np.ndarray[np.integer]` - a NumPy NDArray with *dtype np.integer*.
- `pd.Series[np.integer]` - a Pandas Series with *dtype np.integer*.
- `torch.Tensor[torch.int64]` - a PyTorch Tensor with *dtype torch.int*.
- `mk.AbstractColumn` - a Meerkat column who's cells are `int`, `np.integer`, or `torch.int64`.

This is useful when the rows are neither coontiguous nor evenly spaced (otherwise slice indexing, described above, is faster).

Selecting Multiple Rows from a DataPanel: `Sequence[bool]` -> *DataPanel*

To select multiple rows from a DataPanel we can also pass a list of `bool` the same length as the DataPanel. Below we select the first and last rows from the smaller DataPanel `small_dp` that we selected in the panel above.

```
In [23]: small_dp[[True, False, False, False, True]]
Out[23]: DataPanel(nrows: 2, ncols: 13)
```

Other valid sequences of `bool` that can be used to select are:

- `Tuple[bool]` – a tuple of bool.
- `np.ndarray[bool]` - a NumPy NDArray with *dtype bool*.
- `pd.Series[bool]` - a Pandas Series with *dtype bool*.
- `torch.Tensor[torch.bool]` - a PyTorch Tensor with *dtype torch.bool*.

- `mk.AbstractColumn` - a Meerkat column who's cells are `int`, `bool`, or `torch.bool`.

This is very useful for quickly selecting a subset of rows that satisfy a predicate (like you might do with a `WHERE` clause in SQL). For example, say we want to select all rows that have a value of `"parachute"` in the `"label"` column. We could do this using the following code:

```
In [24]: small_dp.lz[small_dp["label"] == "parachute"]
Out[24]: DataPanel(nrows: 0, ncols: 13)
```

Copy vs. Reference

See *Copy vs. View Behavior* for more information.

You may be wondering whether the rows returned by indexing are copies or references of the rows in the original DataPanel. This depends on (1) which of the selection strategies above you use (`slice` vs. `Sequence[int]` vs. `Sequence[bool]`) and (2) the column type (*e.g.* `PandasSeriesColumn`, `NumpyArrayColumn`).

In general, columns inherit the copying behavior of their underlying data structure. For example, a `NumpyArrayColumn` has the copying behavior of a NumPy array, as described in the Numpy indexing documentation. See a more detailed discussion in *Copy vs. View Behavior*.

For Pandas Users

`.iloc` **and** `.loc`: Pandas users are likely familiar with `.iloc` and `.loc` properties of DataFrames and Series. These properties are used to select data by integer position and by label in the index, respectively.In Meerkat, DataPanels and Columns do **not** have a designated index object as do DataFrames and Series. In Meerkat, the primary way to select rows in Meerkat is by integer position or boolean mask, so there is no need for distinct `.iloc` and `loc` indexers.

Indexing Cells: In Pandas, it's possible to select a cell directly from a DataFrame with a single index like `df.loc[2, "label"]`. This is **not** supported in Meerkat. Instead you should chain the indexing operators together. For example, `dp["label"][2]`. In general, you should index the column first and then the row. Doing it in the reverse order could be wasteful, since the other cells in the row would be loaded for no reason.

Lambda Columns and Lazy Selection

Lambda Columns

If you check out the implementation of *ImageColumn*, you'll notice that it's a super simple subclass of *LambdaColumn*.

What's a LambdaColumn? In Meerkat, high-dimensional data types like images and videos are typically stored in a *LambdaColumn*. A *LambdaColumn* wraps around another column and applies a function to it's content as it is indexed.

Consider the following example, where we create a simple Meerkat column...

```
In [1]: import meerkat as mk

In [2]: col = mk.NumpyArrayColumn([0,1,2])

In [3]: col[1]
Out[3]: 1
```

...and wrap it in a lambda column.

```
In [4]: lambda_col = col.to_lambda(function=lambda x: x + 10)

In [5]: lambda_col[1]  # the function is only called at this point!
Out[5]: 11
```

Critically, the function inside a lambda column is only called at the time the column is indexed! This is very useful for columns with large data types that we don't want to load all into memory at once. For example, we could create a *LambdaColumn* that lazily loads images...

```
In [6]: from PIL import Image

In [7]: dp = mk.DataPanel(
   ...:     {
   ...:         "filepath": ["/abs/path/to/image0.jpg", ...],
   ...:         "image_id": ["image0", ...]
   ...:     }
   ...: )
   ...:

In [8]: dp["image"] = dp["filepath"].to_lambda(fn=Image.open)
```

Notice how we provide an absolute path to the images. This makes the column useable from any working directory. However, using absolute paths is in other ways not ideal: what if we want to share the DataPanel and open it on a different machine? In the section below, we discuss a subclass of *LambdaColumn* that makes it easy to manage filepaths.

FileColumn

As discussed above, `FileColumn`, a simple subclass of *LambdaColumn*.

The *FileColumn* constructor takes an additional argument, `base_dir`, which is the base directory from which all file paths are relative. When `base_dir` is provided, the paths passed to `filepaths` should be relative to `base_dir`:

```
In [9]: from PIL import Image

In [10]: dp = mk.DataPanel(
   ....:     {
   ....:         "filepath": ["image0.jpg", ...],
   ....:         "image_id": ["image0", ...]
   ....:     }
   ....: )
   ....:

In [11]: dp["image"] = mk.FileColumn.from_filepaths(
   ....:     filepaths=dp["filepath"],
   ....:     loader=Image.open,
   ....:     base_dir="/abs/path/to",
   ....: )
   ....:
```

The `base_dir` can then be changed at any time, so if we wanted to share the DataPanel with another user, we could instruct them to reset the base_dir using `dp["image"].base_dir = "/other/users/abs/path/to"`. Introducing this additional step isn't ideal though, so we recommend using the environment variables technique as described below.

Using Environment Variables in `base_dir`

Environment variables in the `base_dir` argument are automatically expanded. For example, if you set the environment variable `MEERKAT_BASE_DIR` to `"/abs/path/to"`, then you can use `dp["image"].base_dir = "$MEERKAT_BASE_DIR/path/to"`. This is ideal for sharing DataPanels between different users and machines.

Note that the Meerkat dataset registry relies heavily on this technique, using a special environment variable `MEERKAT_DATASET_DIR` that points to the `mk.config.datasets.root_dir`.

An *ImageColumn* is a just a *FileColumn* like this one, with a few more bells and whistles!

Lazy Selection

Todo: Fill in this stub.

Overview of Column Types

Todo: Fill in this stub. Envisioning an overview of common column types, and when to use them.

I/O

Writing to Disk

Todo: Fill in this stub.

Reading from Disk

Todo: Fill in this stub.

Importing into Meerkat

Todo: Fill in this stub.

Exporting from Meerkat

Todo: Fill in this stub.

Map, Filter, and Update

Map

Todo: Fill in this stub.

Filter

Todo: Fill in this stub.

Update

Todo: Fill in this stub.

Operations

Merge

Todo: Fill in this stub.

Concat

Todo: Fill in this stub.

GroupBy

Todo: Fill in this stub.

Visualization

Todo: Fill in this stub.

Configuring Meerkat

Several aspects of Meerkat's behavior can be configured by the user. For example, one may wish to change the number of DataPanel rows shown in Jupyter Notebooks.

You can see the current state of the Meerkat configuration with:

```
In [1]: import meerkat as mk

In [2]: mk.config
Out[2]: MeerkatConfig(display=DisplayConfig(max_rows=10, show_images=True, max_image_
↪height=128, max_image_width=128, show_audio=True), datasets=DatasetsConfig(_root_dir='/
↪home/docs/.meerkat/datasets'))
```

Configuring with YAML

To make persistent changes to the configuration, edit the YAML file at *~/.meerkat/config.yaml*. For example, the YAML file below will change the default directory to which datasets are downloaded and increase the max number of rows displayed in Jupyter Notebooks:

```
dataset:
    root_dir: "/path/to/storage"
display:
    max_rows: 20
```

If you would rather keep the YAML file elsewhere, then you can set the environment variable `MEERKAT_CONFIG` to point to the file:

```
export MEERKAT_CONFIG="/path/to/mk/config.yaml"
```

If you're using a conda, you can permanently set this variable for your environment:

```
conda env config vars set MEERKAT_CONFIG="path/to/mk/config.yaml"
conda activate env_name  # need to reactivate the environment
```

Configuring Programmatically

You can also update the config programmatically, though, unlike the YAML method above, these changes will not persist beyond the lifetime of your program.

```
mk.config.datasets.root_dir = "/path/to/storage"
mk.config.public_bucket_name = "mk-test"
```

Patterns and Anti-patterns

Todo: Fill in this stub.

2.1.2 Common Use Cases

Creating Machine Learning Datasets with Meerkat

Todo: Fill in this stub.

Training Models with Meerkat

Todo: Fill in this stub.

Evaluating Models with Meerkat

Todo: Fill in this stub.

Performing Error Analysis with Meerkat

Todo: Fill in this stub.

2.1.3 Advanced Topics

Copy vs. View Behavior

In Meerkat, as in other data structures (*e.g.* NumPy, Pandas), it is important to understand whether or not two variables point to objects that share the same underlying data. If they do, modifying one will affect the other. If they don't, data must be getting copied, which could have implications for efficiency. Consider the following example:

```
>>> import meerkat as mk
>>> col1 = mk.NumpyArrayColumn(np.arange(10))
>>> col2 = col1[:4]
>>> col2[0] = -1
>>> print(col1[0])
```

Is `0` or `-1` printed out?

It turns out that in this case it is `-1` that is printed. This is because `col2` is a "view" of the `col1` array, meaning that the two variables point to objects that share the same underlying data. However, if we were to change the third line to `col2 = col1[np.arange(4)]`, a seemingly inconsequential change, then the underlying data would be copied and it would be `0` that is printed.

In this guide, we will discuss how to know when two variables in Meerkat share underlying data. In general, Meerkat inherits the copy and view behavior of its backend data structures (Numpy Arrays, Pandas Series, Torch Tensors). So, users who are are ' familiar with those libraries should find it straightforward to predict Meerkat's copying and viewing behavior.

We'll begin by defining some terms: coreferences, views and copies. These terms describe the different relationships that could exist between two variables pointing to *AbstractColumn* or *DataPanel* objects. Then, we'll discuss how to know whether indexing a Meerkat data structures will result in a copy, coreference or view.

Copies, Views, and Coreferences

Columns

Let's enumerate the different relationships that could exist between two column variables `col1` and `col2`.

Coreferences - Both variables refer to the same *AbstractColumn* object.

```
>>> col1 is col2
True
```

Of course, in this case, anything changes made to `col1` will also be made to `col2` and vice versa.

Views - The variables refer to different *AbstractColumn* objects (*i.e.* `col1 is not col1`), but modifying the data of `col1` affects `col2` :

1. either because `col1.data` and `col2.data` reference the same object

   ```
   # a. the underlying data variables reference the same object
   >>> col1.data is col2.data
   True
   ```

2. or because `col1.data` is a view of `col2.data` (or vice versa)

```
## For example, if col1.data is np.ndarray
>>> isinstance(col1.data, np.ndarray)
True
# b. the underlying data share memory
>>> col1.data.base is col2.data.base
True
```

- *How are views created?* Views of a column are created in one of two ways:

  1. Implicitly with `col._clone(data=new_data)` where `col.data` shares memory with `new_data`for one of the reasons described above.
  2. Explicitly with `col.view()` which is simply a wrapper around `col._clone`:

     ```
     def view(self):
         return self._clone()
     ```

- *What about other attributes?* (*e.g.* `loader` in an `ImageColumn`) It depends.

  `col1` and `col2` refer to different column objects, so assignment to attributes in `col1` will not affect `col2` (and vice versa):

  ```
  >>> col1.loader = fn1
  >>> col1.loader == col2.loader
  False
  ```

  However, these attributes are not copied! So, stateful changes to the attributes will carry across columns:

  ```
  >>> col1.loader.size = 224
  >>> col2.loader.size == 224
  True
  ```

  If we'd like attributes, we'll have to use "*Deep Copies*".

Copies– The variables refer to different *AbstractColumn* objects (*i.e.* `col1 is not col1`), and modifying the data of `col1` does **not** affect `col2`

In this case, `col1.data` and `[col2.data](http://col2.data)` do not share memory.

- *How are copies created?* Copies of a column are created in one of two ways:

  1. Implicitly with `col._clone(data=new_data)` where `[col.data](http://col.data)` does not share memory with `new_data`.
  2. Explicitly with `col.copy()` which is simply a wrapper around `col._clone`:

     ```
     def copy(self):
         new_data = self._copy_data()
         return self._clone(data=new_data)
     ```

     where `_copy_data` is a backend-specific method that copies the data. For example, if the backend is a Numpy Array, then `_copy_data` will simply `return self.data.copy()`. This is an important point: each column must know how to truly copy it's data.

- *What about other attributes?* (*e.g.* `loader` in an `ImageColumn`) Same as "View" above.

DataPanels

Let's do the same for two DataPanel variables `dp1` and `dp2`.

Coreferences - Both variables refer to the same `DataPanel` object.

```
>>> dp1 is dp2
True
```

Of course, in this case, anything that is done to `dp1` will also be done to `dp2` and vice versa.

Views - The variables refer to different `DataPanel` objects (*i.e.* `dp1 is not dp2`), but some of the columns in `dp1` are coreferences or views of some of the columns in `dp2`

- *How are views created? Views* of a DataPanel are created in one of three ways:
  1. Implicitly with `dp._clone(data=new_data)` where `dp.columns` includes some columns with `new_data`for one of the reasons described above.
  2. Implicitly when a column from one DataPanel is added to another (*e.g.* `dp1["a"] = dp2["b"]`. Behind the scenes,
  3. Explicitly with `dp.view()` which simply calls `col.view()` on all its columns and then passes them `dp._clone(data=view_columns)`
- *What about other attributes?* (*e.g.* `index_column` in an `EntityDataPanel`) It depends.

  `dp1` and `dp2` refer to different column objects, so assignment to attributes in `dp1` will not affect `dp2` (and vice versa):

  ```
  >>> dp1.loader = fn1
  >>> dp1.loader == dp2.loader
  False
  ```

  However, these attributes are not copied! So, stateful changes to the attributes will carry across DataPanels:

  ```
  >>> dp1.loader.size = 224
  >>> dp2.loader.size == 224
  True
  ```

Copies– The variables refer to different `DataPanel` objects (*i.e.* `dp1 is not dp2`), and all of the columns in `dp1` are copies of the the columns in `dp2`

- *How are copies created?* Copies of a column are created in one of two ways.
  1. Implicitly with `col._clone(data=new_data)` where `[col.data](http://col.data)` does not share memory with `new_data`.
  2. Explicitly with `col.copy()` which is simply a wrapper around `col._clone`:

     ```
     def copy(self):
         new_data = self._copy_data()
         return self._clone(data=new_data)
     ```

     where `_copy_data` is a backend-specific method that copies the data. For example, if the backend is a Numpy Array, then `_copy_data` will simply `return self.data.copy()`. This is an important point: each column must know how to truly copy it's data.
- *What about other attributes?* (*e.g.* `index_column` in an `EntityDataPanel`) Same as "View" above.

Behavior when Indexing

Indexing rows

In Meerkat, we select rows by indexing with `int`, `slice` , `Sequence[int]`, or an `np.ndarray` , `torch.Tensor`, `pandas.Series` with an integer or boolean type.

We can select rows from an *AbstractColumn*...

```
col: mk.AbstractColumn = ...
# (1) int -> single value
value: object = col[0]
# (2) slice -> a sub column
new_col: mk.AbstractColumn = col[0:10]
# (3) sequence -> a sub column
new_col: mk.AbstractColumn = col[[0, 4, 6]]
```

... or from a `DataPanel`

```
dp: mk.DataPanel = ...
# (1) int -> dict
row: dict = dp[0]
# (2) slice -> a DataPanel slice
new_dp: mk.DataPanel = dp[0:10]
# (3) sequence -> a DataPanel slice
new_dp: mk.Datapanel = dp[[0, 4, 6]]
```

From a column. When selecting rows from a column `col`, Meerkat takes the following approach:

Step 1. Indexes the underlying data object stored at `[col.data](http://col.data)` (*e.g.* `np.ndarray` or `torch.tensor`) *always* deferring to the copy/view strategy of that data structure. This gives us a new data object, `new_data` which may or may not share memory with with the original `col.data` depending on the strategy of the underlying data structure.

- Copy/View strategies of data structures underlying core Meerkat columns.
  - **torch**

    When accessing the contents of a tensor via indexing, PyTorch follows Numpy behaviors that basic indexing returns views, while advanced indexing returns a copy. Assignment via either basic or advanced indexing is in-place. See more examples in Numpy indexing documentation.

  - **numpy**

    Advanced indexing always returns a copy of the data (contrast with basic slicing that returns a view). (source)

  - **pandas**

    But in pandas, whether you get a view or not depends on the structure of the DataFrame and, if you are trying to modify a slice, the nature of the modification. (source)

  One particularly odd behavior that is worht pointing out is as follows

Step 2. Clones the original column, `col`, and stores the the newly indexed data object, `new_data`, in it (*i.e.* with `col._clone(data=new_data)`.

So, selecting rows from a column `col` returns either a view or a copy, depending on the underlying data structure.

From a DataPanel. When selecting rows from a DataPanel `dp`, Meerkat takes the following approach:

Step 1. Indexes each of the columns using the strategy above.

Note: sometimes this step proceeds in batches according to the BlockManager.

Step 2. Clones the original DataPanel, `dp`, passing the newly indexed columns. This new DataPanel will be:

- either a view of the original `dp`, if any of the indexed columns are views
- or a copy if all of the indexed columns are copies

Indexing columns

In Meerkat, we select columns from a `DataPanel` by either indexing with `str` or a `Sequence[str]` :

```
# (1) `str` -> single column
col: mk.AbstractColumn = dp["col_a"]
# (2) `Sequence[str]` -> multiple columns
dp: mk.DataPanel = dp[["col_a", "col_b"]]
```

When selecting columns from a `DataPanel`, Meerkat **always** returns a coreference to the underlying column(s) – *not* a copy or view.

(1) Indexing a single column (*i.e.* with a `str`) returns the underlying *`AbstractColumn`* object directly. In the example below `col1` and `col2` are coreferences of the same column.

```
# (1) `str` -> single column
>>> col1: mk.AbstractColumn = dp["col_a"]
>>> col2: mk.AbstractColumn = dp["col_a"]
>>> col1 is col2
True
```

(2) Indexing multiple columns (*i.e.* with `Sequence[str]`) returns a view of the `DataPanel` holding coreferences to the columns in the original `DataPanel`. This means the *`AbstractColumn`* objects held in the new `DataPanel` are the same *`AbstractColumn`* objects held in the original `DataPanel`.

```
# (1) `Sequence[str]` -> single column
>>> new_dp: mk.DataPanel = dp[["col_a", "col_b"]]
>>> new_dp["col_a"] is dp["col_a"]
True
>>> new_dp["col_a"].data is dp["col_a"].data
True
```

Subclassing DataPanel and AbstractColumn

Subclassing AbstractColumn

Todo: Fill in this stub.

Subclassing DataPanel

Todo: Fill in this stub.

2.1.4 Meerkat Internals

DataPanel Internals: Blocks

In Meerkat, the columns of a DataPanel are grouped together into *blocks*, sets of columns with similar underlying storage (*e.g.* NumPy arrays). Organizing columns into blocks enables:

1. Vectorized row-wise operations (*e.g.* slicing, reduction)
2. Simplified I/O and improved latency

The most important internal piece of the Meerkat `DataPanel` implementation is the `BlockManager`, a dict-like object that maps column names to columns. The `BlockManager` manages links between a DataPanel's columns and data blocks (*AbstractBlock*, *NumpyBlock*) where the data is actually stored. It implements *consolidate*, which takes columns of similar type in a DataPanel and stores their data together in a block, and *apply* which applies row-wise operations (e.g. __getitem__) to the blocks in a vectorized fashion. Other important classes:

- `BlockRef` objects link a block with the `BlockManager`. These are critical to the functioning of the BlockManager and are the primary type of object passed between the blocks and the block manager. They consists of two things:
  1. A reference to the block (*self.block*)
  2. A set of columns in the `BlockManager` whose data live in the *Block*
- `BlockableMixin` - a mixin used with *AbstractColumn* that holds references to a column's block and the columns index in the block
- `BlockView` - a simple DataClass holding a block and an index into the block. It is typical for new columns to be created from *BlockView*

`BlockManager`

Manages all the columns in a `DataPanel` and holds references (`BlockRef`) to all the blocks in a `DataPanel`. This is done with two collections:

- `_columns`, a dictionary mapping from column names to `AbstractColumn`
- `_block_refs`, a dictionary mapping from the blocks id to `BlockRef`

Implement the following methods:

``consolidate``

```
### PSEUDOCODE
block_groups = group blocks by signature
for group in block_groups:
    for block in group:
        # get a "view" of the subset of the columns in the block
    # (note this may take multiple )
```

(continues on next page)

(continued from previous page)

```
    # concat the blocks and get mapping from name
  # and figure out the mapping of columns to index in block
```

IMPORTANT: After a consolidate, all columns have their own memory!

apply

How do block operations work?

- Apply the operation to each block in the data panel,
  - Each new block should
- Create mapping

add

- Single
- Multiple

remove

When deleting a column we have to be sure to delete the reference to the block ****

get_columns

BlockRef

A BlockRef is the link between a DataPanel and a single block. It consists of two things:

- A reference to the block (self._block)
- A set of columns (of typeBlockableMixin

AbstractBlock

Multiple A block can exist in multiple .

BlockableMixin

This is mixed into AbstractColumn subclasses that can take part of a block (*e.g.*

Meerkat Test Suite

Column Test Beds

Fixtures

2.2 Datasets

Meerkat provides a dataset registry that makes it easy to download datasets and load them into Meerkat data structures. For example, using `get()` we can download and prepare the Imagenette dataset:

```
In [1]: import meerkat as mk

In [2]: dp = mk.datasets.get("imagenette")
```

Some datasets have multiple versions, for example Imagenette provides a full-size version as well as 320 pixel and 160 pixel versions. You can list a dataset's available versions with `versions()`:

```
In [3]: mk.datasets.versions("imagenette")
Out[3]: ['full', '320px', '160px']

In [4]: mk.datasets.get("imagenette", version="160px")
```

By default datasets are downloaded to `~/.meerkat/datasets/{name}/{version}`. However, if you already have the dataset downloaded elsewhere or you want to download to a different location, you can specify the `dataset_dir` argument.

```
dp = mk.datasets.get("imagenette", dataset_dir="/local/download/of/imagenette/full")
```

You can also configure Meerkat to use a different default root directory. By setting the `mk.config.datasets.root_dir = "/local/download/of"`, the default location for datasets will be `/local/download/of/datasets/{name}/{version}`.

How does Meerkat's dataset registry fit in with other dataset hubs? The purpose of the Meerkat dataset registry is to provide *code* for downloading datasets and loading them into *DataPanel* objects. The Meerkat registry, like Torchvision Datasets, doesn't actually host any data. In contrast, dataset hubs like HuggingFace Datasets and Activeloop Hub are great community efforts that *do* host data. So, the Meerkat registry is complementary to these hubs: in fact, we can currently load any dataset in the HuggingFace hubs directly through our registry. For example, we can load the IMBD dataset hosted on HuggingFace with `mk.datasets.get("imdb")`.

Contributing Datasets

We encourage users to contribute datasets to the Meerkat registry. If you're already using Meerkat with your dataset, contributing it to the registry is straightforward: you just share the code that you're already using to load the dataset into Meerkat. Please follow the instructions in *Contributing Datasets*.

The table below lists all of the datasets currently in the meerkat registry. You can also list these datasets programmatically with `mk.datasets.catalog`.

2.2.1 Contributing Datasets

2.3 API Reference

2.3.1 meerkat package

Subpackages

meerkat.block package

Submodules

meerkat.block.abstract module

class AbstractBlock(**args*, ***kwargs*)

Bases: object

classmethod consolidate(*block_refs: Sequence[*BlockRef*]*, *consolidated_inputs: Dict[int, 'AbstractColumn'] = None*) → Tuple[*AbstractBlock*, Mapping[str, BlockIndex]]

classmethod from_block_data(*data: object*) → Tuple[*AbstractBlock*, *BlockView*]

classmethod from_column_data(*data: object*) → Tuple[*AbstractBlock*, *BlockView*]

classmethod read(*path: str*, **args*, ***kwargs*)

write(*path: str*, **args*, ***kwargs*)

property is_mmap

property signature: Hashable

class BlockView(*block_index: 'BlockIndex'*, *block: 'AbstractBlock'*)

Bases: object

block: *AbstractBlock*

block_index: Union[int, slice, str]

property data

meerkat.block.arrow_block module

class ArrowBlock(*data: Table*, **args*, ***kwargs*)

Bases: *AbstractBlock*

class Signature(*nrows: 'int'*, *klass: 'type'*)

Bases: object

klass: type

nrows: int

classmethod from_block_data(*data: Table*) → List[*BlockView*]

classmethod from_column_data(*data: Array*) → *BlockView*

property signature: Hashable

meerkat.block.manager module

class BlockManager

Bases: MutableMapping

Manages all blocks in a DataPanel.

add_column(*col:* AbstractColumn, *name: str*)

Convert data to a meerkat column using the appropriate Column type.

apply(*method_name: str = '_get'*, **args*, ***kwargs*) → *BlockManager*

consolidate(*consolidate_unitary_groups: bool = False*)

copy()

classmethod from_dict(*data: Mapping[str, object]*)

get_block_ref(*name: str*)

classmethod read(*path: str*, *columns: Optional[Sequence[str]] = None*, ***kwargs*) → *BlockManager*

Load a DataPanel stored on disk.

remove(*name*)

reorder(*order: Sequence[str]*)

topological_block_refs()

Topological sort of the block refs based on Kahn's algorithm.

update(*block_ref:* BlockRef)

data (): a single blockable object, potentially contains multiple columns.

view()

write(*path: str*)

property ncols

property nrows

meerkat.block.numpy_block module

class NumpyBlock(*data*, **args*, ***kwargs*)

Bases: *AbstractBlock*

class Signature(*dtype: 'np.dtype'*, *nrows: 'int'*, *shape: 'Tuple[int]'*, *klass: 'type'*, *mmap: 'Union[bool, int]'*)

Bases: object

dtype: dtype

klass: type

mmap: Union[bool, int]

nrows: int

shape: Tuple[int]

classmethod from_column_data(*data: ndarray*) → Tuple[*NumpyBlock*, *BlockView*]

[summary]

Parameters

- **data** (*np.ndarray*) – [description]
- **names** (*Sequence[str]*) – [description]

Raises

ValueError – [description]

Returns

[description]

Return type

Tuple[*NumpyBlock*, Mapping[str, BlockIndex]]

property is_mmap

property signature: Hashable

meerkat.block.pandas_block module

class PandasBlock(*data: DataFrame*, **args*, ***kwargs*)

Bases: *AbstractBlock*

class Signature(*nrows: 'int'*, *klass: 'type'*)

Bases: object

klass: type

nrows: int

classmethod from_column_data(*data: Series*) → Tuple[*PandasBlock*, *BlockView*]

[summary]

Parameters

- **data** (*np.ndarray*) – [description]
- **names** (*Sequence[str]*) – [description]

Raises

ValueError – [description]

Returns

[description]

Return type

Tuple[*PandasBlock*, Mapping[str, BlockIndex]]

property signature: Hashable

meerkat.block.ref module

class BlockRef(*columns: Mapping[str,* AbstractColumn*]*, *block:* AbstractBlock)

Bases: Mapping

apply(*method_name: str = '_get'*, **args*, ***kwargs*) → Union[*BlockRef*, List[*BlockRef*], dict]

update(*block_ref:* BlockRef)

property block_indices

meerkat.block.tensor_block module

class TensorBlock(*data*, **args*, ***kwargs*)

Bases: *AbstractBlock*

class Signature(*device: 'torch.device'*, *dtype: 'torch.dtype'*, *nrows: 'int'*, *shape: 'Tuple[int]'*, *klass: 'type'*)

Bases: object

device: device

dtype: dtype

klass: type

nrows: int

shape: Tuple[int]

classmethod from_column_data(*data: Tensor*) → Tuple[*TensorBlock*, *BlockView*]

[summary]

Parameters

- **data** (*np.ndarray*) – [description]
- **names** (*Sequence[str]*) – [description]

Raises

ValueError – [description]

Returns

[description]

Return type

Tuple[*NumpyBlock*, Mapping[str, BlockIndex]]

property signature: Hashable

Module contents

meerkat.cells package

Submodules

meerkat.cells.abstract module

class AbstractCell(**args*, ***kwargs*)

Bases: ABC

get(**args*, ***kwargs*) → object

Get me the thing that this cell exists for.

loader(**args*, ***kwargs*) → object

property metadata: dict

Get the metadata associated with this cell.

meerkat.cells.spacy module

class LazySpacyCell(*text: str*, *nlp: spacy.language.Language*, **args*, ***kwargs*)

Bases: *AbstractCell*

default_loader(**args*, ***kwargs*)

classmethod from_state(*state*, *nlp: spacy.language.Language*)

get(**args*, ***kwargs*)

Get me the thing that this cell exists for.

get_state()

class SpacyCell(*doc: spacy_tokens.Doc*, **args*, ***kwargs*)

Bases: *AbstractCell*

default_loader(**args*, ***kwargs*)

classmethod from_state(*state*, *nlp: spacy.language.Language*)

get(**args*, ***kwargs*)

Get me the thing that this cell exists for.

get_state()

meerkat.cells.volume module

class MedicalVolumeCell(*paths: Union[str, Path, PathLike, Sequence[Union[str, Path, PathLike]]]*, *loader: Optional[Callable] = None*, *transform: Optional[Callable] = None*, *cache_metadata: bool = False*, **args*, ***kwargs*)

Bases: *PathsMixin*, *AbstractCell*

Interface for loading medical volume data.

Examples

Specify xray dicoms with default orientation `("SI", "AP")`: `>>>` cell = MedicalVolumeCell("/path/to/xray.dcm", loader=DicomReader(group_by=None, default_ornt=("SI", "AP"))

Load multi-echo MRI volumes `>>>` cell = MedicalVolumeCell("/path/to/mri/scan/dir", loader=DicomReader(group_by="EchoNumbers"))

clear_metadata()

classmethod default_loader(*paths: Sequence[Path]*, **args*, ***kwargs*)

classmethod from_state(*state*, **args*, ***kwargs*)

get(**args*, *cache_metadata: Optional[bool] = None*, ***kwargs*)

Get me the thing that this cell exists for.

get_metadata(*ignore_bytes: bool = False*, *readable: bool = False*, *as_raw_type: bool = False*, *force_load: bool = False*) → Dict

get_state()

Module contents

meerkat.columns package

Submodules

meerkat.columns.abstract module

class AbstractColumn(*data: Optional[Sequence] = None*, *collate_fn: Optional[Callable] = None*, *formatter: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: *BlockableMixin*, *CloneableMixin*, *CollateMixin*, *ColumnIOMixin*, *FunctionInspectorMixin*, *LambdaMixin*, *MappableMixin*, *MaterializationMixin*, *ProvenanceMixin*, ABC

An abstract class for Meerkat columns.

append(*column:* AbstractColumn) → None

argsort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *AbstractColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
AbstractColumn

batch(*batch_size: int = 1*, *drop_last_batch: bool = False*, *collate: bool = True*, *num_workers: int = 0*, *materialize: bool = True*, **args*, ***kwargs*)

Batch the column.

Parameters

- **batch_size** – integer batch size
- **drop_last_batch** – drop the last batch if its smaller than batch_size
- **collate** – whether to collate the returned batches

Returns
batches of data

static concat(*columns: Sequence[*AbstractColumn*]*) → None

filter(*function: Callable*, *with_indices=False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: Optional[int] = 0*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → Optional[*AbstractColumn*]

Filter the elements of the column using a function.

classmethod from_data(*data: Union[Columnable,* AbstractColumn*]*)

Convert data to a meerkat column using the appropriate Column type.

full_length()

classmethod get_writer(*mmap: bool = False*, *template: Optional[*AbstractColumn*] = None*)

head(*n: int = 5*) → *AbstractColumn*

Get the first *n* examples of the column.

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

sample(*n: Optional[int] = None*, *frac: Optional[float] = None*, *replace: bool = False*, *weights: Optional[Union[str, ndarray]] = None*, *random_state: Optional[Union[int, RandomState]] = None*) → *AbstractColumn*

Select a random sample of rows from Column. Roughly equivalent to `sample` in Pandas https://pandas.pydata.org/docs/reference/api/panda s.DataFrame.sample.html.

Parameters

- **n** (*int*) – Number of samples to draw. If *frac* is specified, this parameter should not be passed. Defaults to 1 if *frac* is not passed.

- **frac** (*float*) – Fraction of rows to sample. If *n* is specified, this parameter should not be passed.
- **replace** (*bool*) – Sample with or without replacement. Defaults to False.
- **weights** (*np.ndarray*) – Weights to use for sampling. If *None* (default), the rows will be sampled uniformly. If a numpy array, the sample will be weighted accordingly. If weights do not sum to 1 they will be normalized to sum to 1.
- **random_state** (*Union[int, np.random.RandomState]*) – Random state or seed to use for sampling.

Returns
A random sample of rows from the DataPanel.

Return type
AbstractColumn

sort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *AbstractColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
AbstractColumn

streamlit()

tail(*n: int = 5*) → *AbstractColumn*

Get the last *n* examples of the column.

to_pandas() → Series

Columnable

alias of Union[Sequence, ndarray, Series, Tensor]

property data

Get the underlying data.

property formatter: Callable

property is_mmap

logdir: Path = PosixPath('/home/docs/meerkat')

property metadata

meerkat.columns.arrow_column module

class ArrowArrayColumn(*data: Sequence*, **args*, ***kwargs*)

Bases: *AbstractColumn*

block_class

alias of *ArrowBlock*

classmethod concat(*columns: Sequence[*ArrowArrayColumn*]*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (*AbstractColumn*) – [description]

to_numpy()

to_pandas()

to_tensor()

meerkat.columns.audio_column module

class AudioColumn(*data: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*, **args*, ***kwargs*)

Bases: *FileColumn*

A lambda column where each cell represents an audio file on disk. The underlying data is a *PandasSeriesColumn* of strings, where each string is the path to an image. The column materializes the images into memory when indexed. If the column is lazy indexed with the `lz` indexer, the images are not materialized and an `FileCell` or an `AudioColumn` is returned instead.

Parameters

- **data** (*Sequence[str]*) – A list of filepaths to images.
- **transform** (*callable*) – A function that transforms the image (e.g. `torchvision.transforms.functional.center_crop`).

  Warning: In order for the column to be serializable, the transform function must be pickleable.

- **loader** (*callable*) – A callable with signature `def loader(filepath: str) -> PIL.Image:`. Defaults to `torchvision.datasets.folder.default_loader`.

  Warning: In order for the column to be serializable with `write()`, the loader function must be pickleable.

- **base_dir** (*str*) – A base directory that the paths in `data` are relative to. If `None`, the paths are assumed to be absolute.

collate(*batch*)

Collate data.

classmethod default_loader(**args*, ***kwargs*)

meerkat.columns.cell_column module

class CellColumn(*cells: Optional[Sequence[*AbstractCell*]] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*

static concat(*columns: Sequence[*CellColumn*]*)

classmethod from_cells(*cells: Sequence[*AbstractCell*]*, **args*, ***kwargs*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

property cells

meerkat.columns.file_column module

class Downloader(*cache_dir: str*, *downloader: Optional[callable] = None*)

Bases: object

class FileCell(*data: LambdaCellOp*)

Bases: *LambdaCell*

from_filepath(*transform: Optional[callable] = None*, *loader: Optional[callable] = None*, *path: Optional[str] = None*, *base_dir: Optional[str] = None*)

property absolute_path

class FileColumn(*data: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*, **args*, ***kwargs*)

Bases: *LambdaColumn*

A column where each cell represents an file stored on disk or the web. The underlying data is a *PandasSeriesColumn* of strings, where each string is the path to a file. The column materializes the files into memory when indexed. If the column is lazy indexed with the `lz` indexer, the files are not materialized and a `FileCell` or a `FileColumn` is returned instead.

Parameters

- **data** (*Sequence[str]*) – A list of filepaths to images.
- **transform** (*callable*) – A function that transforms the image (e.g. `torchvision.transforms.functional.center_crop`).

> **Warning:** In order for the column to be serializable, the transform function must be pickleable.

- **loader** (*callable*) – A callable with signature `def loader(filepath: str) -> PIL.Image:`. Defaults to `torchvision.datasets.folder.default_loader`.

  Warning: In order for the column to be serializable with `write()`, the loader function must be pickleable.

- **base_dir** (*str*) – A base directory that the paths in `data` are relative to. If `None`, the paths are assumed to be absolute.

classmethod default_loader(**args*, ***kwargs*)

classmethod from_filepaths(*filepaths: Sequence[str]*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

property base_dir

property loader

property transform

class FileLoader(*transform: Optional[callable] = None*, *loader: Optional[callable] = None*, *base_dir: Optional[str] = None*)

Bases: `object`

download_image(*url: str*, *cache_dir: str*)

meerkat.columns.image_column module

class ImageColumn(*data: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*, **args*, ***kwargs*)

Bases: *FileColumn*

A column where each cell represents an image stored on disk. The underlying data is a *PandasSeriesColumn* of strings, where each string is the path to an image. The column materializes the images into memory when indexed. If the column is lazy indexed with the `lz` indexer, the images are not materialized and an `ImageCell` or an `ImageColumn` is returned instead.

Parameters

- **data** (*Sequence[str]*) – A list of filepaths to images.
- **transform** (*callable*) – A function that transforms the image (e.g. `torchvision.transforms.functional.center_crop`).

  Warning: In order for the column to be serializable, the transform function must be pickleable.

- **loader** (*callable*) – A callable with signature `def loader(filepath: str) -> PIL.Image:`. Defaults to `torchvision.datasets.folder.default_loader`.

  Warning: In order for the column to be serializable with `write()`, the loader function must be pickleable.

- **base_dir** (*str*) – A base directory that the paths in `data` are relative to. If `None`, the paths are assumed to be absolute.

classmethod default_loader(**args*, ***kwargs*)

meerkat.columns.lambda_column module

class LambdaCell(*data: LambdaCellOp*)

Bases: *AbstractCell*

get(**args*, ***kwargs*)

Get me the thing that this cell exists for.

property data: object

Get the data associated with this cell.

class LambdaColumn(*data: Union[LambdaOp,* BlockView*]*, *output_type: Optional[type] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*

block_class

alias of `LambdaBlock`

static concat(*columns: Sequence[*LambdaColumn*]*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (*AbstractColumn*) – [description]

property fn: Callable

Subclasses like *ImageColumn* should be able to implement their own version.

meerkat.columns.list_column module

class ListColumn(*data: Optional[Sequence] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*

batch(*batch_size: int = 1*, *drop_last_batch: bool = False*, *collate: bool = True*, **args*, ***kwargs*)

Batch the column.

Parameters

- **batch_size** – integer batch size
- **drop_last_batch** – drop the last batch if its smaller than batch_size
- **collate** – whether to collate the returned batches

Returns
batches of data

classmethod concat(*columns: Sequence[*ListColumn*]*)

default_formatter()

classmethod from_list(*data: Sequence*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

meerkat.columns.numpy_column module

class NumpyArrayColumn(*data: Sequence*, **args*, ***kwargs*)

Bases: *AbstractColumn*, NDArrayOperatorsMixin

block_class

alias of *NumpyBlock*

argsort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *NumpyArrayColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
NumpySeriesColumn

For now! Raises error when shape of input array is more than one error.

classmethod concat(*columns: Sequence[*NumpyArrayColumn*]*)

classmethod from_array(*data: ndarray*, **args*, ***kwargs*)

classmethod from_npy(*path*, *mmap_mode=None*, *allow_pickle=False*, *fix_imports=True*, *encoding='ASCII'*)

classmethod get_writer(*mmap: bool = False*, *template: Optional[*AbstractColumn*] = None*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

sort(*ascending: Union[bool, List[bool]] = True*, *axis: int = -1*, *kind: str = 'quicksort'*, *order: Optional[Union[str, List[str]]] = None*) → *NumpyArrayColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
AbstractColumn

to_numpy() → ndarray

to_pandas() → Series

to_tensor() → Tensor

Use *column.to_tensor()* instead of *torch.tensor(column)*, which is very slow.

property is_mmap

getattr_decorator(*fn: Callable*)

meerkat.columns.pandas_column module

class PandasSeriesColumn(*data: Optional[Sequence] = None*, *collate_fn: Optional[Callable] = None*, *formatter: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*, NDArrayOperatorsMixin

block_class

alias of *PandasBlock*

cat

alias of _MeerkatCategoricalAccessor

dt

alias of _MeerkatCombinedDatetimelikeProperties

str

alias of _MeerkatStringMethods

argsort(*ascending: bool = True*, *kind: str = 'quicksort'*) → *PandasSeriesColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type

PandasSeriesColumn

For now! Raises error when shape of input array is more than one error.

classmethod concat(*columns: Sequence[PandasSeriesColumn]*)

classmethod from_array(*data: ndarray*, **args*, ***kwargs*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters

other (AbstractColumn) – [description]

sort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *PandasSeriesColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

AbstractColumn

to_numpy() → Tensor

to_pandas() → Series

to_tensor() → Tensor

Use *column.to_tensor()* instead of *torch.tensor(column)*, which is very slow.

getattr_decorator(*fn: Callable*)

meerkat.columns.spacy_column module

class SpacyColumn(*data: Sequence[spacy_tokens.Doc] = None*, **args*, ***kwargs*)

Bases: ListColumn

classmethod from_docs(*data: Sequence[spacy_tokens.Doc]*, **args*, ***kwargs*)

classmethod from_texts(*texts: Sequence[str]*, *lang: str = 'en_core_web_sm'*, **args*, ***kwargs*)

classmethod read(*path: str*, *nlp: spacy.language.Language = None*, *lang: str = None*, **args*, ***kwargs*) → *SpacyColumn*

write(*path: str*, ***kwargs*) → None

property docs

property tokens

meerkat.columns.tensor_column module

class TensorColumn(*data: Optional[Sequence] = None*, **args*, ***kwargs*)

Bases: NDArrayOperatorsMixin, *AbstractColumn*

block_class

alias of *TensorBlock*

argsort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *TensorColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

TensorColumn

For now! Raises error when shape of input array is more than one error.

classmethod concat(*columns: Sequence[*TensorColumn*]*)

classmethod from_data(*data: Union[Sequence, ndarray, Series, Tensor,* AbstractColumn*]*)

Convert data to an EmbeddingColumn.

classmethod get_writer(*mmap: bool = False*, *template: Optional[*AbstractColumn*] = None*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters

other (AbstractColumn) – [description]

sort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *TensorColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

AbstractColumn

to_numpy() → Series

to_pandas() → Series

to_tensor() → Tensor

getattr_decorator(*fn: Callable*)

meerkat.columns.volume_column module

class MedicalVolumeColumn(**args*, ***kwargs*)

Bases: *CellColumn*

classmethod from_filepaths(*filepaths: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, **args*, ***kwargs*)

Module contents

meerkat.datasets package

Subpackages

meerkat.datasets.audioset package

Module contents

build_audioset_dp(*dataset_dir: str*, *splits: Optional[List[str]] = None*, *audio_column: bool = True*, *overwrite: bool = False*) → Dict[str, *DataPanel*]

Build DataPanels for the audioset dataset downloaded to `dataset_dir`. By default, the resulting DataPanels will be written to `dataset_dir` under the filenames "audioset_examples.mk" and "audioset_labels.mk". If these files already exist and `overwrite` is False, the DataPanels will not be built anew, and instead will be simply loaded from disk.

Parameters

- **dataset_dir** – The directory where the dataset is stored
- **download** – Whether to download the dataset
- **splits** – A list of splits to include. Defaults to ["eval_segments"]. Other splits: "balanced_train_segments", "unbalanced_train_segments".
- **audio_column** (*bool*) – Whether to include a *AudioColumn*. Defaults to True.
- **overwrite** (*bool*) – Whether to overwrite existing DataPanels saved to disk. Defaults to False.

build_ontology_dp(*dataset_dir: str*) → Dict[str, *DataPanel*]

Build a DataPanel from the ontology.json file.

Parameters

dataset_dir – The directory where the ontology.json file is stored

find_submids(*id: Union[List[str], str]*, *relations: Optional[*DataPanel*] = None*, *dataset_dir: Optional[str] = None*) → List[str]

Returns a list of IDs of all subcategories of an audio category.

Parameters

- **ids** – ID or list of IDs for which to find the subcategories
- **dp** – A DataPanel built from the ontology.json file.
- **dataset_dir** – Alternatively, the directory where the ontology.json file is stored can be provided to construct a DataPanel

meerkat.datasets.celeba package

Module contents

class celeba(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['main']

info: *DatasetInfo* = DatasetInfo(name='celeba', full_name='CelebFaces Attributes', description='CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 attribute annotations. The images in this dataset cover large pose variations and background clutter.', citation=None, homepage='https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html', license=None, tags=['image', 'face recognition'])

build_celeba_df(*dataset_dir: str*)

Build the dataframe by joining on the attribute, split and identity CelebA CSVs.

download_celeba(*dataset_dir: str*)

get_celeba(*dataset_dir: str*, *download: bool = False*)

Build the dataframe by joining on the attribute, split and identity CelebA CSVs.

meerkat.datasets.coco package

Module contents

class coco(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['2014']

info: *DatasetInfo* = DatasetInfo(name='coco', full_name='Common Objects in Context', description='Image data sets for object class recognition.', citation=None, homepage='https://cocodataset.org/#home', license=None, tags=['image', 'object recognition'])

build_coco_2014_dp(*dataset_dir: str*, *download: bool = False*)

meerkat.datasets.dew package

Module contents

build_dew_dp(*dataset_dir: str*, *download: bool = True*) → *DataPanel*

meerkat.datasets.eeg package

Submodules

meerkat.datasets.eeg.data_utils module

Module contents

meerkat.datasets.enron package

Module contents

build_enron_dp(*dataset_dir: str*, *download: bool = True*) → *DataPanel*

meerkat.datasets.gqa package

Module contents

build_gqa_dps(*dataset_dir: str*, *write: bool = False*) → Dict[str, *DataPanel*]

crop_object(*row: Mapping[str, object]*)

read_gqa_dps(*dataset_dir: str*) → Dict[str, *DataPanel*]

write_gqa_dps(*dps: Mapping[str,* DataPanel*]*, *dataset_dir: str*)

meerkat.datasets.imagenet package

Module contents

class imagenet(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['ilsvrc2012']

info: *DatasetInfo* = DatasetInfo(name='imagenet', full_name='ImageNet', description='ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images..', citation='@inproceedings{imagenet_cvpr09,AUTHOR = {Deng, J. and Dong, W. and Socher, R. and Li, L.-J. and Li, K. and Fei-Fei, L.},TITLE = {{ImageNet: A Large-Scale Hierarchical Image Database}},BOOKTITLE = {CVPR09},YEAR = {2009},BIBSOURCE = "http://www.image-net.org/papers/imagenet_cvpr09.bib"}', homepage='https://www.image-net.org/', license=None, tags=['image', 'classification'])

build_imagenet_dps(*dataset_dir: str*, *download: bool = False*) → Dict[str, *DataPanel*]

meerkat.datasets.imagenette package

Module contents

class imagenette(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['full', '320px', '160px']

VERSION_TO_URL = {'160px': 'https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.tgz', '320px': 'https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-320.tgz', 'full': 'https://s3.amazonaws.com/fast-ai-imageclas/imagenette2.tgz'}

property data_dir

info: *DatasetInfo* = DatasetInfo(name='imagenette', full_name='ImageNette', description='Imagenette is a subset of 10 easily classified classes from Imagenet (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute).', citation=None, homepage='https://github.com/fastai/imagenette', license=None, tags=['image', 'classification'])

build_imagenette_dp(*dataset_dir: str*, *download: bool = False*, *version: str = '160px'*) → *DataPanel*

Build DataPanel for the Imagenette dataset.

Parameters

- **download_dir** (*str*) – The directory path to save to or load from.
- **version** (*str, optional*) – Imagenette version. Choices: `"full"`, `"320px"`, `"160px"`.
- **overwrite** (*bool, optional*) – If `True`, redownload the datasets.

Returns

A DataPanel corresponding to the dataset.

Return type

mk.DataPanel

References

https://github.com/fastai/imagenette

download_imagenette(*download_dir*, *version='160px'*, *overwrite: bool = False*, *return_df: bool = False*)

Download Imagenette dataset.

Parameters

- **download_dir** (*str*) – The directory path to save to.
- **version** (*str, optional*) – Imagenette version. Choices: `"full"`, `"320px"`, `"160px"`.
- **overwrite** (*bool, optional*) – If `True`, redownload the dataset.
- **return_df** (*bool, optional*) – If `True`, return a `pd.DataFrame`.

Returns

If `return_df=True`, returns a pandas DataFrame.

Otherwise, returns the directory path where the data is stored.

Return type

Union[str, pd.DataFrame]

References

https://github.com/fastai/imagenette

meerkat.datasets.inaturalist package

Module contents

build_inaturalist_dp(*dataset_dir: str*, *download: bool = True*, *splits: Optional[List[str]] = None*) → *DataPanel*

Build a DataPanel from the inaturalist dataset.

Parameters

- **dataset_dir** – The directory to store the dataset in.
- **download** – Whether to download the dataset if it does not yet exist.
- **splits** – A list of splits to include. Defaults to all splits.

meerkat.datasets.mimic package

Submodules

meerkat.datasets.mimic.gcs module

meerkat.datasets.mimic.modules module

meerkat.datasets.mimic.reports module

Module contents

meerkat.datasets.mir package

Module contents

meerkat.datasets.pascal package

Module contents

class pascal(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['2012']

VERSION_TO_URL = {'2012':
'http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCtrainval_11-May-2012.tar'}

```
info: DatasetInfo = DatasetInfo(name='pascal', full_name='PASCAL',
description='Image data sets for object class recognition.',
citation='@Article{Everingham10,author = "Everingham, M. and Van~Gool, L. and
Williams, C. K. I. and Winn,J. and Zisserman, A.",title = "The Pascal Visual Object
Classes (VOC) Challenge",journal = "International Journal of Computer Vision",volume
= "88",year = "2010",number = "2",month = jun,pages = "303--338",}',
homepage='http://host.robots.ox.ac.uk/pascal/VOC/', license=None, tags=['image',
'object recognition'])
```

build_pascal_2012_dp(*dataset_dir: str*)

meerkat.datasets.siim_cxr package

Module contents

cxr_transform(*volume:* MedicalVolumeCell)

cxr_transform_pil(*volume:* MedicalVolumeCell)

download_siim_cxr(*dataset_dir: str*, *kaggle_username: str*, *kaggle_key: str*, *download_gaze_data: bool = True*, *include_mock_reports: bool = True*)

Download the dataset from the SIIM-ACR Pneumothorax Segmentation challenge. https://www.kaggle.com/c/siim-acr-pneumothorax- segmentation/data.

Parameters

- **dataset_dir** (*str*) – Path to directory where the dataset will be downloaded.
- **kaggle_username** (*str*) – Your kaggle username.
- **kaggle_key** (*str*) – A kaggle API key. In order to use the Kaggle's public API, you must first authenticate using an API token. From the site header, click on your user profile picture, then on "My Account" from the dropdown menu. This will take you to your account settings at https://www.kaggle.com/account. Scroll down to the section of the page labelled API: To create a new token, click on the "Create New API Token" button. This will download a json file with a "username" and "key" field. Copy and paste the "key" field and pass it in as *kaggle_key*. Instructions copied from Kaggle API docs: https://www.kaggle.com/docs/api
- **download_gaze_data** (*str*) – Download a pkl file containing eye-tracking data collected on a radiologist interpreting the xray.

meerkat.datasets.torchaudio package

Module contents

meerkat.datasets.torchvision package

Module contents

get_cifar10(*download_dir: str*, *frac_val: float = 0.0*, *download: bool = True*)

Load CIFAR10 as a Meerkat DataPanel.

Parameters

- **download_dir** – download directory
- **frac_val** – fraction of training set to use for validation

Returns
a DataPanel containing columns *raw_image*, *image* and *label*

get_torchvision_dataset(*dataset_name*, *download_dir*, *is_train*)

meerkat.datasets.video_corruptions package

Submodules

meerkat.datasets.video_corruptions.transforms module

class TemporalCrop(*n_clips: int*, *clip_length: int*, *time_dim: Optional[int] = 1*, *clip_spacing: Optional[str] = 'equal'*, *padding_mode: Optional[str] = 'loop'*, *sample_starting_location: Optional[bool] = False*, *stack_clips: Optional[bool] = True*)

Bases: `object`

Video transformation for performing "temporal cropping:" the sampling of a pre-defined number of clips, each with pre-defined length, from a full video. Can be used with Compose in torchvision.

When used with TemporalDownsampling, it is highly recommended to put TemporalCrop after TemporalDownsampling. Since TemporalCrop can change the number of dimensions in the output tensor, due to clip selection, it is in fact recommended to put this transform at the end of a video transformation pipeline.

Parameters

- **n_clips** (*int*) – the number of clips that should be sampled.
- **clip_length** (*int*) – the length of each clip (in the number of frames)
- **time_dim** (*int*) – the index of the time dimension of the video
- **clip_spacing** (*Optional; default "equal"*) – how to choose starting locations for sampling clips. Keyword "equal" means that clip starting locations are sampled from each 1/n_clips segment of the video. The other option, "anywhere", places no restrictions on where clip starting locations can be sampled.
- **padding_mode** – (Optional; default "loop"): behavior if a requested clip length would result a clip exceeding the end of the video. Keyword "loop" results in a wrap-around to the start of the video. The other option, "freeze", repeats the final frame until the requested clip length is achieved.
- **sample_starting_location** – (Optional; default False): whether to sample a starting location (usually used for training) for a clip. Can be used in tandem with "equal" during training to sample clips with random starting locations distributed across time. Redundant if *clip_spacing* is "anywhere".
- **stack_clips** – (Optional; default True): whether to stack clips in a new dimension (used in 3D action recognition backbones), or stack clips by concatenating across the time dimension (used in 2D action recognition backbones). Output shape if True is (n_clips, *video_shape). If False, the output shape has the same number of dimensions as the original video, but the time dimension is extended by a factor of n_clips.

Examples

Create a VideoCell from "/path/to/video.mp4" with time in dimension one, sampling 10 clips each of length 16, sampling clips equally across the video >>> cell = VideoCell("/path/to/video.mp4",

time_dim=1, transform=TemporalCrop(10, 16, time_dim=1))

output shape: (10, n_channels, 16, H, W)

Create a VideoCell from "/path/to/video.mp4" with time in dimension one, sampling 8 clips each of length 8, sampling clips from arbitrary video locations and freezing the last frame if a clip exceeds the video length >>> cell = VideoCell("/path/to/video.mp4",

time_dim=1, transform=TemporalCrop(8, 8, time_dim=1, clip_spacing="anywhere", padding_mode="freeze"))

output shape: (8, n_channels, 8, H, W)

Create a VideoCell from "/path/to/video.mp4" with time in dimension one, sampling one frame from each third of the video, concatenating the frames in the time dimension >>> cell = VideoCell("/path/to/video.mp4",

time_dim=1, transform=TemporalCrop(1, 3, time_dim=1, clip_spacing="equal",

sample_starting_location=True, stack_clips=False)

)

output shape: (n_channels, 3, H, W)

Note that time_dim in the TemporalDownsampling call must match the the time_dim in the VideoCell constructor!

class TemporalDownsampling(*downsample_factor: int*, *time_dim: Optional[int] = 1*)

Bases: `object`

Video transformation for performing temporal downsampling (i.e. reading in every Nth frame only). This can be used in tandem with VideoCell by passing it into the *transform* keyword in the constructor. Can be used with Compose in torchvision.

When using with TemporalCrop, it is highly recommended to put TemporalDownsampling first, with TemporalCrop second.

Parameters

- **downsample_factor** (*int*) – the factor by which the input video should be downsampled. Must be a strictly positive integer.
- **time_dim** (*int*) – the time dimension of the input video.

Examples

Create a VideoCell from "/path/to/video.mp4" with time in dimension one, showing every other frame >>> cell = VideoCell("path/to/video.mp4",

time_dim=1, transform=TemporalDownsampling(2, time_dim=1))

Note that time_dim in the TemporalDownsampling call must match the the time_dim in the VideoCell constructor!

meerkat.datasets.video_corruptions.utils module

class stderr_suppress

Bases: object

A context manager for doing a "deep suppression" of stdout and stderr in Python.

This is necessary when reading in a corrupted video, or else stderr will emit 10000s of errors via ffmpeg. Great for decoding IRL, not great for loading 100s of corrupted videos.

Module contents

meerkat.datasets.visual_genome package

Module contents

build_visual_genome_dps(*dataset_dir: str*, *write: bool = False*) → Dict[str, *DataPanel*]

read_visual_genome_dps(*dataset_dir: str*) → Dict[str, *DataPanel*]

write_visual_genome_dps(*dps: Mapping[str,* DataPanel*]*, *dataset_dir: str*)

meerkat.datasets.waterbirds package

Module contents

meerkat.datasets.wilds package

Submodules

meerkat.datasets.wilds.config module

WILDS configuration defaults and operations.

All default configurations are integrated from the WILDS repository: https://github.com/p-lambda/wilds/blob/ca7e4fd345aa4f998b8322191b083444d85e2a08/examples/configs/datasets.py

populate_config(*config*, *template: dict*, *force_compatibility=False*)

Populates missing (key, val) pairs in config with (key, val) in template. Example usage: populate config with defaults :param - config: namespace :param - template: dict :param - force_compatibility: option to raise errors if config.key != template[key]

populate_defaults(*config*)

Populates hyperparameters with defaults implied by choices of other hyperparameters.

meerkat.datasets.wilds.transforms module

getBertTokenizer(*model*)

initialize_bert_transform(*config*)

initialize_image_base_transform(*config*, *dataset*)

initialize_image_resize_and_center_crop_transform(*config*, *dataset*)

Resizes the image to a slightly larger square then crops the center.

initialize_poverty_train_transform()

initialize_transform(*transform_name*, *config*, *dataset*)

Module contents

WILDS integration for Meerkat.

class WILDSInputColumn(*dataset_name: str = 'fmow'*, *version: Optional[str] = None*, *root_dir: Optional[str] = None*, *split: Optional[str] = None*, *use_transform: bool = True*, ***kwargs*)

Bases: *AbstractColumn*

get_metadata_columns()

get_y_column()

Get a NumpyArrayColumn holding the targets for the dataset.

Warning: *WildsDataset`s may remap indexes in arbitrary ways so it's important not to directly try to access the underlying data structures, instead relying on the `y_array* and *metadata_array* properties which are universal across WILDS datasets.

get_wilds_datapanel(*dataset_name: str*, *root_dir: str*, *version: Optional[str] = None*, *column_names: Optional[List[str]] = None*, *info: Optional[*DatasetInfo*] = None*, *split: Optional[str] = None*, *use_transform: bool = True*, *include_raw_input: bool = True*)

Get a DataPanel that holds a WildsInputColumn alongside NumpyColumns for targets and metadata.

Example: Run inference on the dataset and store predictions alongside the data. .. code-block:: python

dp = get_wilds_datapanel("fmow", root_dir="/datasets/", split="test") model = … # get the model model.to(0).eval()

@torch.no_grad() def predict(batch: dict):

out = torch.softmax(model(batch["input"].to(0)), axis=-1) return {"pred": out.cpu().numpy().argmax(axis=-1)}

dp = dp.update(function=predict, batch_size=128, is_batched_fn=True)

Parameters

- **dataset_name** (*str, optional*) – dataset name. Defaults to *"fmow"*.
- **version** (*str, optional*) – dataset version number, e.g., '1.0'. Defaults to the latest version.
- **root_dir** (*str*) – the directory where the WILDS dataset is downloaded. See https://wilds.stanford.edu/ for download instructions.
- **split** (*str, optional*) – see . Defaults to None.

- **use_transform** (*bool, optional*) – Whether to apply the transform from the WILDS example directory on load. Defaults to True.
- **column_names** (*List[str], optional*) – [description]. Defaults to None.
- **info** (DatasetInfo, *optional*) – [description]. Defaults to None.
- **use_transform** – [description]. Defaults to True.
- **include_raw_input** (*bool, optional*) – include a column for the input without the transform applied – useful for visualizing images. Defaults to True.

Submodules

meerkat.datasets.abstract module

class DatasetBuilder(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: ABC

abstract build()

abstract download()

download_url(*url: str*)

dump_download_meta()

is_downloaded() → bool

This is a very weak check for the existence of the dataset.

Subclasses should ideally implement more thorough checks.

REVISIONS: List[str]

info: *DatasetInfo* = None

meerkat.datasets.fsdd module

meerkat.datasets.info module

class DatasetInfo(*name: str*, *full_name: str = None*, *description: str = None*, *citation: str = None*, *homepage: str = None*, *license: str = None*, *tags: str = None*)

Bases: object

citation: str = None

description: str = None

full_name: str = None

homepage: str = None

license: str = None

name: str

tags: str = None

meerkat.datasets.registry module

class Registry(*name: str*)

Bases: Registry

Extension of fvcore's registry that supports aliases.

get(*name: str*, ***kwargs*) → Any

get_obj(*name: str*) → type

register(*obj: Optional[object] = None*, *aliases: Optional[Sequence[str]] = None*) → Optional[object]

Register the given object under the the name *obj.__name__*. Can be used as either a decorator or not. See docstring of this class for usage.

property catalog: *DataPanel*

property names: List[str]

meerkat.datasets.utils module

download_google_drive(*url: Optional[str] = None*, *id: Optional[str] = None*, *dst: Optional[str] = None*, *is_folder: bool = False*)

download_url(*url: str*, *dataset_dir: str*, *force: bool = False*)

extract(*path: str*, *dst: str*, *extractor: Optional[str] = None*)

Module contents

class celeba(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['main']

info: *DatasetInfo* = DatasetInfo(name='celeba', full_name='CelebFaces Attributes', description='CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than 200K celebrity images, each with 40 attribute annotations. The images in this dataset cover large pose variations and background clutter.', citation=None, homepage='https://mmlab.ie.cuhk.edu.hk/projects/CelebA.html', license=None, tags=['image', 'face recognition'])

class coco(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['2014']

info: DatasetInfo = DatasetInfo(name='coco', full_name='Common Objects in Context', description='Image data sets for object class recognition.', citation=None, homepage='https://cocodataset.org/#home', license=None, tags=['image', 'object recognition'])

class expw(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: DatasetBuilder

build()

download()

REVISIONS: List[str]

VERSIONS = ['main']

VERSION_TO_GDRIVE_ID = {'main': '19Eb_WiTsWelYv7Faff0L5Lmo1zv0vzwR'}

info: DatasetInfo = DatasetInfo(name='expw', full_name='Expression in-the-Wild', description='Imagenette is a subset of 10 easily classified classes from Imagenet (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute).', citation=None, homepage='https://github.com/fastai/imagenette', license=None, tags=['image', 'classification'])

class fer(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: DatasetBuilder

build()

download()

REVISIONS: List[str]

VERSIONS = ['plus']

info: DatasetInfo = DatasetInfo(name='fer', full_name='Facial Expression Recognition Challenge', description='ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images..', citation=None, homepage='https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge/data?select=icml_face_data.csv', license=None, tags=['image', 'facial emotion recognition'])

class imagenet(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: DatasetBuilder

build()

download()

REVISIONS: List[str]

VERSIONS = ['ilsvrc2012']

info: *DatasetInfo* = DatasetInfo(name='imagenet', full_name='ImageNet', description='ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images..', citation='@inproceedings{imagenet_cvpr09,AUTHOR = {Deng, J. and Dong, W. and Socher, R. and Li, L.-J. and Li, K. and Fei-Fei, L.},TITLE = {{ImageNet: A Large-Scale Hierarchical Image Database}},BOOKTITLE = {CVPR09},YEAR = {2009},BIBSOURCE = "http://www.image-net.org/papers/imagenet_cvpr09.bib"}', homepage='https://www.image-net.org/', license=None, tags=['image', 'classification'])

class imagenette(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['full', '320px', '160px']

VERSION_TO_URL = {'160px': 'https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-160.tgz', '320px': 'https://s3.amazonaws.com/fast-ai-imageclas/imagenette2-320.tgz', 'full': 'https://s3.amazonaws.com/fast-ai-imageclas/imagenette2.tgz'}

property data_dir

info: *DatasetInfo* = DatasetInfo(name='imagenette', full_name='ImageNette', description='Imagenette is a subset of 10 easily classified classes from Imagenet (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute).', citation=None, homepage='https://github.com/fastai/imagenette', license=None, tags=['image', 'classification'])

class mirflickr(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build() → *DataPanel*

download()

REVISIONS: List[str]

VERSIONS = ['25k']

VERSION_TO_URLS = {'25k':
['http://press.liacs.nl/mirflickr/mirflickr25k.v3b/mirflickr25k.zip', 'http://press.
liacs.nl/mirflickr/mirflickr25k.v3b/mirflickr25k_annotations_v080.zip']}

info: *DatasetInfo* = DatasetInfo(name='mirflickr', full_name='PASCAL',
description='The MIRFLICKR-25000 open evaluation project consists of 25000 images
downloaded from the social photography site Flickr through its public API coupled
with complete manual annotations, pre-computed descriptors and software for
bag-of-words based similarity and classification and a matlab-like tool for
exploring and classifying imagery.', citation="@inproceedings{huiskes08, author =
{Mark J. Huiskes and Michael S. Lew}, title = {The MIR Flickr Retrieval Evaluation},
booktitle = {MIR '08: Proceedings of the 2008 ACM International Conference on
Multimedia Information Retrieval}, year = {2008}, location = {Vancouver, Canada},
publisher = {ACM}, address = {New York, NY, USA},}",
homepage='https://press.liacs.nl/mirflickr/', license=None, tags=['image',
'retrieval'])

class ngoa(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

class Downloader(*cache_dir: str*, *downloader: Optional[callable] = None*)

Bases: object

build()

download()

REVISIONS: List[str]

VERSIONS = ['main']

info: *DatasetInfo* = DatasetInfo(name='ngoa', full_name='National Gallery of Art Open
Data', description='The dataset provides data records relating to the 130,000+
artworks in our collection and the artists who created them. You can download the
dataset free of charge without seeking authorization from the National Gallery of
Art.', citation=None, homepage='https://github.com/NationalGalleryOfArt/opendata',
license=None, tags=['art'])

class pascal(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

REVISIONS: List[str]

VERSIONS = ['2012']

VERSION_TO_URL = {'2012':
'http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCtrainval_11-May-2012.tar'}

info: *DatasetInfo* = DatasetInfo(name='pascal', full_name='PASCAL', description='Image data sets for object class recognition.', citation='@Article{Everingham10,author = "Everingham, M. and Van~Gool, L. and Williams, C. K. I. and Winn,J. and Zisserman, A.",title = "The Pascal Visual Object Classes (VOC) Challenge",journal = "International Journal of Computer Vision",volume = "88",year = "2010",number = "2",month = jun,pages = "303--338",}', homepage='http://host.robots.ox.ac.uk/pascal/VOC/', license=None, tags=['image', 'object recognition'])

class rfw(*dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, ***kwargs*)

Bases: *DatasetBuilder*

build()

download()

GROUPS = ['Caucasian', 'African', 'Asian', 'Indian']

REVISIONS: List[str]

VERSIONS = ['main']

info: *DatasetInfo* = DatasetInfo(name='fer', full_name='Racial Faces in-the-Wild', description='Racial Faces in-the-Wild (RFW) is a testing database for studying racial bias in face recognition. Four testing subsets, namely Caucasian, Asian, Indian and African, are constructed, and each contains about 3000 individuals with 6000 image pairs for face verification. They can be used to fairly evaluate and compare the recognition ability of the algorithm on different races.', citation=None, homepage='http://www.whdeng.cn/RFW/testing.html', license=None, tags=['image', 'facial recognition', 'algorithmic bias'])

meerkat.logging package

Submodules

meerkat.logging.utils module

initialize_logging(*log_dir: Optional[str] = None*, *log_name: str = 'meerkat.log'*, *format: str = '[%(asctime)s][%(levelname)s][%(name)s:%(lineno)s] :: %(message)s'*, *level: int = 30*) → None

Initialize logging for Meerkat.

set_logging_level(*level: Union[int, str] = 20*)

Set logging level for Meerkat.

set_logging_level_for_imports(*level: int = 30*) → None

Set logging levels for dependencies.

Module contents

meerkat.mixins package

Submodules

meerkat.mixins.blockable module

class BlockableMixin(**args*, ***kwargs*)

Bases: object

classmethod is_blockable()

block_class: type = None

meerkat.mixins.cloneable module

class CloneableMixin(**args*, ***kwargs*)

Bases: object

copy(***kwargs*) → object

view() → object

class StateClass(*klass: type*, *state: object*)

Bases: object

An internal class to store the state of an object alongside its associated class.

klass: type

state: object

meerkat.mixins.collate module

class CollateMixin(*collate_fn: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: object

collate(**args*, ***kwargs*)

Collate data.

property collate_fn

Method used to collate.

identity_collate(*batch: List*)

meerkat.mixins.file module

class FileMixin(*filepath: Union[str, Path]*, **args*, ***kwargs*)

Bases: object

Mixin for adding in single filepath.

class PathsMixin(*paths: Union[str, Path, PathLike, Sequence[Union[str, Path, PathLike]]]*, **args*, ***kwargs*)

Bases: object

Mixin for adding in generic paths.

meerkat.mixins.inspect_fn module

class FunctionInspectorMixin(**args*, ***kwargs*)

Bases: object

meerkat.mixins.io module

class ColumnIOMixin

Bases: object

classmethod read(*path: str*, *_data: Optional[object] = None*, *_meta: Optional[object] = None*, **args*, ***kwargs*) → object

write(*path: str*, **args*, ***kwargs*) → None

meerkat.mixins.lambdable module

class LambdaMixin(**args*, ***kwargs*)

Bases: object

to_lambda(*function: Callable*, *is_batched_fn: bool = False*, *batch_size: int = 1*, *inputs: Union[Mapping[str, str], Sequence[str]] = None*, *outputs: Union[Mapping[any, str], Sequence[str]] = None*, *output_type: Union[Mapping[str, type], type] = None*) → Union[*DataPanel*, *LambdaColumn*]

summary

Examples

Parameters

- **self** –
- **function** (*Callable*) – The function that will be applied to the rows of self.
- **is_batched_fn** (*bool, optional*) – Whether the function must be applied on a batch of rows. Defaults to False.
- **batch_size** (*int, optional*) – The minimum batch size . Ignored if is_batched_fn=False. Defaults to 1.

- **inputs** (*Dict[str, str], optional*) – Dictionary mapping column names in `self` to keyword arguments of `function`. Ignored if `self` is a column. When calling `function` values from the columns will be fed to the corresponding keyword arguments. Defaults to None, in which case the entire datapanel.
- **outputs** (*Union[Dict[any, str], Tuple[str]], optional*) – Controls how the output of `function` is mapped to the returned `LambdaColumn`(s). Defaults to None. * If ``None`, a single `LambdaColumn` is returned. * If a `Dict[any, str]`, then a `DataPanel` containing `LambdaColumn`s is returned. This is useful when the output of ``function` is a `Dict`. `outputs` maps the outputs of `function` to column names in the resulting `DataPanel`. * If a `Tuple[str]`, then a `DataPanel` containing `LambdaColumn`s is returned. , This is useful when the output of ``function` is a `Tuple`. `outputs` maps the outputs of `function` to column names in the resulting `DataPanel`.
- **output_type** (*Union[Dict[str, type], type], optional*) – _description_. Defaults to None.

Raises
ValueError – _description_

Returns
description

Return type
type

to_lambda(*data: Union[DataPanel, AbstractColumn]*, *function: Callable*, *is_batched_fn: bool = False*, *batch_size: int = 1*, *inputs: Union[Mapping[str, str], Sequence[str]] = None*, *outputs: Union[Mapping[any, str], Sequence[str]] = None*, *output_type: Union[Mapping[str, type], type] = None*) → Union[*DataPanel*, *LambdaColumn*]

summary

Examples

Parameters

- **data** –
- **function** (*Callable*) – The function that will be applied to the rows of `data`.
- **is_batched_fn** (*bool, optional*) – Whether the function must be applied on a batch of rows. Defaults to False.
- **batch_size** (*int, optional*) – The minimum batch size . Ignored if `is_batched_fn=False`. Defaults to 1.
- **inputs** (*Dict[str, str], optional*) – Dictionary mapping column names in `data` to keyword arguments of `function`. Ignored if `data` is a column. When calling `function` values from the columns will be fed to the corresponding keyword arguments. Defaults to None, in which case the entire datapanel.
- **outputs** (*Union[Dict[any, str], Tuple[str]], optional*) – Controls how the output of `function` is mapped to the returned `LambdaColumn`(s). Defaults to None. * If ``None`, a single `LambdaColumn` is returned. * If a `Dict[any, str]`, then a `DataPanel` containing `LambdaColumn`s is returned. This is useful when the output of ``function` is a `Dict`. `outputs` maps the outputs of `function` to

column names in the resulting DataPanel. * If a Tuple[str], then a DataPanel containing LambdaColumn`s is returned. , This is useful when the output of ``function` is a Tuple. outputs maps the outputs of function to column names in the resulting DataPanel.

- **output_type** (*Union[Dict[str, type], type], optional*) – _description_. Defaults to None.

Raises
ValueError – _description_

Returns
description

Return type
type

meerkat.mixins.mapping module

class MappableMixin(**args*, ***kwargs*)

Bases: object

map(*function: Optional[Callable] = None*, *with_indices: bool = False*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: Optional[int] = 0*, *output_type: Union[type, Dict[str, type]] = None*, *materialize: bool = True*, *pbar: bool = False*, *mmap: bool = False*, *mmap_path: str = None*, *flush_size: int = None*, ***kwargs*)

meerkat.mixins.materialize module

class MaterializationMixin(**args*, ***kwargs*)

Bases: object

property lz

Module contents

meerkat.ml package

Submodules

meerkat.ml.activation module

meerkat.ml.callbacks module

meerkat.ml.embedding_column module

class EmbeddingColumn(*data: Optional[Sequence] = None*, **args*, ***kwargs*)

Bases: *TensorColumn*

build_faiss_index(*index=None*, *overwrite=False*)

pca(*n_components=2*)

search(*query*, *k: int*)

umap(*n_neighbors=15*, *n_components=2*)

visualize_umap(*n_neighbors=15*, *n_components=2*, *point_size=4*)

meerkat.ml.huggingfacemodel module

meerkat.ml.instances_column module

meerkat.ml.metrics module

accuracy(*predictions: Union[list, array, Tensor]*, *labels: Union[list, array, Tensor]*)

Calculate accuracy.

class_distribution(*labels: Union[list, array, Tensor]*, *num_classes: Optional[int] = None*, *min_label: int = 0*)

Calculate the aggregated class distribution.

compute_metric(*metric: str*, *predictions: Union[Sequence, Tensor]*, *labels: Union[Sequence, Tensor]*, *num_classes: int*) → Union[float, ndarray, Tensor]

Compute metric given predictions and target labels.

Parameters

- **metric** (*str*) – name of metric
- **predictions** (*Union[Sequence, torch.Tensor]*) – a sequence of predictions (rouge metrics) or a torch Tensor (other metrics) containing predictions
- **labels** (*Union[Sequence, torch.Tensor]*) – a sequence of labels (rouge metrics) or a torch Tensor (other metrics) containing target labels
- **num_classes** (*int*) – number of classes

Returns

the calculate metric value

dice(*predictions: Union[list, array, Tensor]*, *labels: Union[list, array, Tensor]*)

Calculate Dice Score.

f1(*predictions: Union[list, array, Tensor]*, *labels: Union[list, array, Tensor]*)

Calculate F1 score for binary classification.

f1_macro(*predictions: Union[list, array, Tensor]*, *labels: Union[list, array, Tensor]*)

Calculate macro F1 score for multi-class classification.

f1_micro(*predictions: Union[list, array, Tensor]*, *labels: Union[list, array, Tensor]*)

Calculate micro F1 score for multi-class classification.

format_summary(*x: str*) → str

Format summary text for computing rouge.

get_metric(*name: str*) → Callable

Get metrics from string names.

iou_score(*predictions: Union[list, array, Tensor]*, *labels: Union[list, array, Tensor]*, *num_classes: Optional[int] = None*)

Calculate IoU.

meerkat.ml.model module

class Model(*model: Module*, *is_classifier: Optional[bool] = None*, *task: Optional[str] = None*, *device: Optional[str] = None*)

Bases: `Module`

activation(*dataset:* DataPanel, *target_module: str*, *input_columns: List[str]*, *batch_size=32*) → *EmbeddingColumn*

An Operation that stores model activations in a new Embedding column.

Parameters

- **dataset** (DataPanel) – the meerkat DataPanel containing the model inputs.
- **target_module** (*str*) – the name of the submodule of *model* (i.e. an intermediate layer) that outputs the activations we'd like to extract. For nested submodules, specify a path separated by "." (e.g. *ActivationCachedOp(model, "block4.conv")*).
- **input_columns** (*str*) – Column containing model inputs

classification(*dataset:* DataPanel, *input_columns: List[str]*, *batch_size: int = 32*, *num_classes: Optional[int] = None*, *multi_label: bool = False*, *one_hot: Optional[bool] = None*, *threshold=0.5*) → *DataPanel*

evaluate(*dataset:* DataPanel, *target_column: List[str]*, *pred_column: List[str]*, *metrics: List[str]*, *num_classes: Optional[int] = None*)

forward(*input_batch: Dict*) → Dict

Defines the computation performed at every call.

Should be overridden by all subclasses.

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

---

static remap_labels(*output_dict: Dict*, *label_map: List[int]*) → Dict

Map the output labels of the model.

Example: 3-way classificaiton, with label_map = [1, 2, 0] => (model label 0 -> dataset label 1, model label 1 -> dataset label 2, ...).

training: bool

meerkat.ml.prediction_column module

class ClassificationOutputColumn(*logits: Optional[Union[Sequence, ndarray, Series, Tensor]] = None*, *probs: Optional[Union[Sequence, ndarray, Series, Tensor]] = None*, *preds: Optional[Union[Sequence, ndarray, Series, Tensor]] = None*, *num_classes: Optional[int] = None*, *multi_label: bool = False*, *one_hot: Optional[bool] = None*, *threshold=0.5*, **args*, ***kwargs*)

Bases: *TensorColumn*

bincount() → *TensorColumn*

Compute the count (cardinality) for each category.

Categories which are not available will have a count of 0.

If `self.multi_label=True`, the bincount will include the total number of times the category is seen. If an example is marked as 2 categories, the bincount will increase the count for both categories. Note, this means the sum of the number of classes can be more than the number of examples `N`.

Returns
A 1D tensor of length `self.num_classes`.

Return type
torch.Tensor

entropy() → *TensorColumn*

Compute the entropy for each example.

If `self.multi_label` is True, each category is treated as a binary classification problem. There will be an entropy calculation for each category as well. For example, if the probabilities are of shape `(N, C)`, there will be `NxC` entropy values.

In the multi-dimensional case, this returns the entropy for each element. For example, if the probabilities are of shape `(N, C, A, B)`, there will be `NxAxB` entropy values.

Returns
Tensor of entropies

Return type
TensorColumn

logits() → *ClassificationOutputColumn*

mode()

predictions() → *ClassificationOutputColumn*

Compute predictions.

preds() → *ClassificationOutputColumn*

Compute predictions.

probabilities() → *ClassificationOutputColumn*

probs() → *ClassificationOutputColumn*

meerkat.ml.segmentation_column module

meerkat.ml.tensormodel module

Module contents

meerkat.ops package

Submodules

meerkat.ops.concat module

concat(*objs: Union[Sequence[*DataPanel*], Sequence[*AbstractColumn*]]*, *axis: Union[str, int] = 'rows'*, *suffixes: Tuple[str] = None*, *overwrite: bool = False*) → Union[*DataPanel*, *AbstractColumn*]

Concatenate a sequence of columns or a sequence of *DataPanel`s. If sequence is empty, returns an empty `DataPanel*.

- If concatenating columns, all columns must be of the same type (e.g. all

ListColumn). - If concatenating `DataPanel`s along axis 0 (rows), all `DataPanel`s must have the same set of columns. - If concatenating `DataPanel`s along axis 1 (columns), all `DataPanel`s must have the same length and cannot have any of the same column names.

Parameters

- **objs** (*Union[Sequence[DataPanel], Sequence[AbstractColumn]]*) – sequence of columns or DataPanels.
- **axis** (*Union[str, int]*) – The axis along which to concatenate. Ignored if concatenating columns.

Returns
concatenated DataPanel or column

Return type
Union[*DataPanel*, *AbstractColumn*]

meerkat.ops.groupby module

class GroupBy(*data:* DataPanel, *indices: Dict[Union[str, Tuple[str]], ndarray]*, *by: Union[List[str], str]*)

Bases: `object`

mean(**args*, ***kwargs*)

groupby(*data:* DataPanel, *by: Optional[Union[str, Sequence[str]]] = None*) → *GroupBy*

Perform a groupby operation on a DataPanel or Column (similar to a *DataFrame.groupby* and *Series.groupby* operations in Pandas).

TODO (Sam): I put down a very rough scaffolding of how you could setup the class hierarchy for this. It is inspired by the way pandas has things setup: check out https://github.com/pandas-dev/pandas/tree/a8968bfa696d51f73769c54f2630a9530488236a/pandas/core/groupby for some inspiration.

I'd recommend starting with small simple datapanels. e.g. a datapanel of all numpy array columns. For example, ``` dp = DataPanel({

'a': NumpyArrayColumn([1, 2, 2, 1, 3, 2, 3]), 'b': NumpyArrayColumn([1, 2, 3, 4, 5, 6, 7]), 'c': NumpyArrayColumn([1.0, 3.2, 2.1, 4.3, 5.4, 6.5, 7.6])

})

groupby(dp, by="a")["c"].mean() ```

Eventually we'll want to support a bunch of different aggregations, but for the time being let's just focus on mean, sum, and count.

Note: we'll also want to implement methods *DataPanel.groupby* or *AbstractColumn.groupby* eventually, but we also want a functional version

that could be called like *mk.groupby(dp, by="class")*. I'd suggest putting most of the implementation here,

and then making the methods just wrappers. See merge as an example.

Parameters

- **data** (*Union[DataPanel, AbstractColumn]*) – The data to group.
- **by** (*Union[str, Sequence[str]]*) – The column(s) to group by. Ignored if `data` is a Column.

Returns
A GroupBy object.

Return type
Union[DataPanelGroupBy, AbstractColumnGroupBy]

meerkat.ops.merge module

merge(*left: DataPanel*, *right: DataPanel*, *how: str = 'inner'*, *on: Union[str, List[str]] = None*, *left_on: Union[str, List[str]] = None*, *right_on: Union[str, List[str]] = None*, *sort: bool = False*, *suffixes: Sequence[str] = ('_x', '_y')*, *validate=None*)

Module contents

meerkat.pipelines package

Submodules

meerkat.pipelines.entitydatapanel module

Module contents

meerkat.tools package

Submodules

meerkat.tools.lazy_loader module

Lazy loader class.

class LazyLoader(*local_name*, *parent_module_globals=None*, *warning=None*, *error=None*)

Bases: module

Lazily import a module, mainly to avoid pulling in large dependencies.

contrib, and *ffmpeg* are examples of modules that are large and not always needed, and this allows them to only be loaded when they are used.

meerkat.tools.utils module

class MeerkatLoader(*stream*)

Bases: FullLoader

PyYaml does not load unimported modules for safety reasons.

We want to allow importing only meerkat modules

find_python_module(*name: str*, *mark*, *unsafe=False*)

find_python_name(*name: str*, *mark*, *unsafe=False*)

convert_to_batch_column_fn(*function: Callable*, *with_indices: bool*, *materialize: bool = True*, ***kwargs*)

Batch a function that applies to an example.

convert_to_batch_fn(*function: Callable*, *with_indices: bool*, *materialize: bool = True*, ***kwargs*)

Batch a function that applies to an example.

nested_getattr(*obj*, *attr*, **args*)

Get a nested property from an object.

noqa: E501 Source: https://stackoverflow.com/questions/31174295/getattr-and-setattr-on-nested-subobjects-chained-properties

translate_index(*index*, *length: int*)

Module contents

meerkat.writers package

Submodules

meerkat.writers.abstract module

class AbstractWriter(**args*, ***kwargs*)

Bases: ABC

abstract close(**args*, ***kwargs*) → None

abstract finalize(**args*, ***kwargs*) → None

abstract flush(**args*, ***kwargs*) → None

abstract open(**args*, ***kwargs*) → None

abstract write(*data*, **args*, ***kwargs*) → None

meerkat.writers.concat_writer module

class ConcatWriter(*output_type: type = <class 'meerkat.columns.abstract.AbstractColumn'>*, *template: ~typing.Optional[~meerkat.columns.abstract.AbstractColumn] = None*, **args*, ***kwargs*)

Bases: *AbstractWriter*

close(**args*, ***kwargs*)

finalize(**args*, ***kwargs*) → None

flush()

open() → None

write(*data*, ***kwargs*) → None

meerkat.writers.numpy_writer module

class NumpyMemmapWriter(*path: ~typing.Optional[str] = None*, *dtype: str = 'float32'*, *mode: str = 'r'*, *shape: ~typing.Optional[tuple] = None*, *output_type: type = <class 'meerkat.columns.numpy_column.NumpyArrayColumn'>*, *template: ~typing.Optional[~meerkat.columns.abstract.AbstractColumn] = None*, **args*, ***kwargs*)

Bases: *AbstractWriter*

close(**args*, ***kwargs*) → None

finalize(**args*, ***kwargs*) → *AbstractColumn*

flush()

Close the mmap file and reopen to release memory.

open(*path: str*, *dtype: str = 'float32'*, *mode: str = 'w+'*, *shape: Optional[tuple] = None*) → None

write(*arr*, ***kwargs*) → None

Module contents

Submodules

meerkat.config module

class DatasetsConfig(*_root_dir: 'str' = '/home/docs/.meerkat/datasets'*)

Bases: object

property root_dir

class DisplayConfig(*max_rows: 'int' = 10*, *show_images: 'bool' = True*, *max_image_height: 'int' = 128*, *max_image_width: 'int' = 128*, *show_audio: 'bool' = True*)

Bases: object

max_image_height: int = 128

max_image_width: int = 128

max_rows: int = 10

show_audio: bool = True

show_images: bool = True

class MeerkatConfig(*display: 'DisplayConfig'*, *datasets: 'DatasetsConfig'*)

Bases: `object`

classmethod from_yaml(*path: Optional[str] = None*)

datasets: *DatasetsConfig*

display: *DisplayConfig*

meerkat.datapanel module

DataPanel class.

class DataPanel(*data: Optional[Union[dict, list]] = None*, **args*, ***kwargs*)

Bases: *CloneableMixin*, *FunctionInspectorMixin*, *LambdaMixin*, *MappableMixin*, *MaterializationMixin*, *ProvenanceMixin*

Meerkat DataPanel class.

add_column(*name: str*, *data: Union[Sequence, ndarray, Series, Tensor]*, *overwrite=False*) → None

Add a column to the DataPanel.

append(*dp:* DataPanel, *axis: Union[str, int] = 'rows'*, *suffixes: Tuple[str] = None*, *overwrite: bool = False*) → *DataPanel*

Append a batch of data to the dataset.

example_or_batch must have the same columns as the dataset (regardless of what columns are visible).

batch(*batch_size: int = 1*, *drop_last_batch: bool = False*, *num_workers: int = 0*, *materialize: bool = True*, *shuffle: bool = False*, **args*, ***kwargs*)

Batch the dataset. TODO:

Parameters

- **batch_size** – integer batch size
- **drop_last_batch** – drop the last batch if its smaller than batch_size

Returns

batches of data

consolidate()

filter(*function: Optional[Callable] = None*, *with_indices=False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: int = 0*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → Optional[*DataPanel*]

Filter operation on the DataPanel.

classmethod from_arrow(*table: Table*)

Create a Dataset from a pandas DataFrame.

classmethod from_batch(*batch: Dict[str, Union[List, AbstractColumn]]*) → *DataPanel*

Convert a batch to a Dataset.

classmethod from_batches(*batches: Sequence[Dict[str, Union[List, AbstractColumn]]]*) → *DataPanel*

Convert a list of batches to a dataset.

classmethod from_csv(*filepath: str*, **args*, ***kwargs*)

Create a Dataset from a csv file.

Parameters

- **filepath** (*str*) – The file path or buffer to load from. Same as `pandas.read_csv()`.
- ***args** – Argument list for `pandas.read_csv()`.
- ****kwargs** – Keyword arguments for `pandas.read_csv()`.

Returns

The constructed datapanel.

Return type

DataPanel

classmethod from_dict(*d: Dict*) → *DataPanel*

Convert a dictionary to a dataset.

Alias for Dataset.from_batch(..).

classmethod from_feather(*path: str*)

Create a Dataset from a feather file.

classmethod from_huggingface(**args*, ***kwargs*)

Load a Huggingface dataset as a DataPanel.

Use this to replace *datasets.load_dataset*, so

```
>>> dict_of_datasets = datasets.load_dataset('boolq')
```

becomes

```
>>> dict_of_datapanels = DataPanel.from_huggingface('boolq')
```

classmethod from_jsonl(*json_path: str*) → *DataPanel*

Load a dataset from a .jsonl file on disk, where each line of the json file consists of a single example.

classmethod from_pandas(*df: DataFrame*)

Create a Dataset from a pandas DataFrame.

groupby(**args*, ***kwargs*)

head(*n: int = 5*) → *DataPanel*

Get the first *n* examples of the DataPanel.

items()

keys()

map(*function: Optional[Callable] = None*, *with_indices: bool = False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: int = 0*, *output_type: Union[type, Dict[str, type]] = None*, *mmap: bool = False*, *mmap_path: str = None*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → Optional[Union[Dict, List, *AbstractColumn*]]

mean(**args*, ***kwargs*) → *DataPanel*

merge(*right:* DataPanel, *how: str = 'inner'*, *on: Optional[Union[str, List[str]]] = None*, *left_on: Optional[Union[str, List[str]]] = None*, *right_on: Optional[Union[str, List[str]]] = None*, *sort: bool = False*, *suffixes: Sequence[str] = ('_x', '_y')*, *validate=None*)

classmethod read(*path: str*, **args*, ***kwargs*) → *DataPanel*

Load a DataPanel stored on disk.

remove_column(*column: str*) → None

Remove a column from the dataset.

sample(*n: Optional[int] = None*, *frac: Optional[float] = None*, *replace: bool = False*, *weights: Optional[Union[str, ndarray]] = None*, *random_state: Optional[Union[int, RandomState]] = None*) → *DataPanel*

Select a random sample of rows from DataPanel. Roughly equivalent to `sample` in Pandas https://pandas.pydata.org/docs/reference/api/panda s.DataFrame.sample.html.

Parameters

- **n** (*int*) – Number of samples to draw. If *frac* is specified, this parameter should not be passed. Defaults to 1 if *frac* is not passed.
- **frac** (*float*) – Fraction of rows to sample. If *n* is specified, this parameter should not be passed.
- **replace** (*bool*) – Sample with or without replacement. Defaults to False.
- **weights** (*Union[str, np.ndarray]*) – Weights to use for sampling. If *None* (default), the rows will be sampled uniformly. If a numpy array, the sample will be weighted accordingly. If a string, the weights will be applied to the rows based on the column with the name specified. If weights do not sum to 1 they will be normalized to sum to 1.
- **random_state** (*Union[int, np.random.RandomState]*) – Random state or seed to use for sampling.

Returns
A random sample of rows from the DataPanel.

Return type
DataPanel

sort(*by: Union[str, List[str]]*, *ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *DataPanel*

Sort the DataPanel by the values in the specified columns. Similar to `sort_values` in pandas.

Parameters

- **by** (*Union[str, List[str]]*) – The columns to sort by.
- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*.Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to ‘quicksort’. Options include ‘quicksort’, ‘mergesort’, ‘heapsort’, ‘stable’.

Returns
A sorted view of DataPanel.

Return type
DataPanel

streamlit()

tail(*n: int = 5*) → *DataPanel*

Get the last *n* examples of the DataPanel.

to_jsonl(*path: str*) → None

Save a Dataset to a jsonl file.

to_pandas() → DataFrame

Convert a Dataset to a pandas DataFrame.

update(*function: Optional[Callable] = None*, *with_indices: bool = False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *remove_columns: Optional[List[str]] = None*, *num_workers: int = 0*, *output_type: Union[type, Dict[str, type]] = None*, *mmap: bool = False*, *mmap_path: str = None*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → *DataPanel*

Update the columns of the dataset.

values()

write(*path: str*) → None

Save a DataPanel to disk.

property columns

Column names in the DataPanel.

property data: *BlockManager*

Get the underlying data (excluding invisible rows).

To access underlying data with invisible rows, use *_data*.

logdir: Path = PosixPath('/home/docs/meerkat')

property ncols

Number of rows in the DataPanel.

property nrows

Number of rows in the DataPanel.

property shape

Shape of the DataPanel (num_rows, num_columns).

meerkat.display module

audio_file_formatter(*cell:* FileCell) → str

auto_formatter(*cell: Any*)

image_file_formatter(*cell:* FileCell)

image_formatter(*cell: Image*)

lambda_cell_formatter(*cell:* LambdaCell)

meerkat.errors module

exception ConcatError
Bases: ValueError

exception ConcatWarning
Bases: RuntimeWarning

exception ConsolidationError
Bases: ValueError

exception ExperimentalWarning
Bases: FutureWarning

exception ImmutableError
Bases: ValueError

exception MergeError
Bases: ValueError

meerkat.provenance module

class ProvenanceMixin(**args*, ***kwargs*)
Bases: object

get_provenance(*include_columns: bool = False*, *last_parent_only: bool = False*)

property node

class ProvenanceNode
Bases: object

add_child(*node:* ProvenanceNode, *key: Tuple*)

add_parent(*node:* ProvenanceNode, *key: Tuple*)

cache_repr()

get_provenance(*last_parent_only: bool = False*)

property children

property last_parent

property parents

class ProvenanceObjNode(*obj:* ProvenanceMixin)
Bases: *ProvenanceNode*

class ProvenanceOpNode(*fn: callable*, *inputs: dict*, *outputs: object*, *captured_args: dict*)
Bases: *ProvenanceNode*

class provenance(*enabled: bool = True*)
Bases: object

capture_provenance(*capture_args: Optional[Sequence[str]] = None*)

get_nested_objs(*data*)

Recursively get DataPanels and Columns from nested collections.

is_provenance_enabled()

set_provenance(*enabled=True*)

visualize_provenance(*obj: Union[ProvenanceObjNode, ProvenanceOpNode]*, *show_columns: bool = False*, *last_parent_only: bool = False*)

meerkat.version module

Module contents

Meerkat.

class AbstractCell(**args*, ***kwargs*)

Bases: ABC

get(**args*, ***kwargs*) → object

Get me the thing that this cell exists for.

loader(**args*, ***kwargs*) → object

property metadata: dict

Get the metadata associated with this cell.

class AbstractColumn(*data: Optional[Sequence] = None*, *collate_fn: Optional[Callable] = None*, *formatter: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: *BlockableMixin*, *CloneableMixin*, *CollateMixin*, *ColumnIOMixin*, *FunctionInspectorMixin*, *LambdaMixin*, *MappableMixin*, *MaterializationMixin*, *ProvenanceMixin*, ABC

An abstract class for Meerkat columns.

append(*column:* AbstractColumn) → None

argsort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *AbstractColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

AbstractColumn

batch(*batch_size: int = 1*, *drop_last_batch: bool = False*, *collate: bool = True*, *num_workers: int = 0*, *materialize: bool = True*, **args*, ***kwargs*)

Batch the column.

Parameters

- **batch_size** – integer batch size
- **drop_last_batch** – drop the last batch if its smaller than batch_size
- **collate** – whether to collate the returned batches

Returns
batches of data

static concat(*columns: Sequence[AbstractColumn]*) → None

filter(*function: Callable*, *with_indices=False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: Optional[int] = 0*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → Optional[*AbstractColumn*]

Filter the elements of the column using a function.

classmethod from_data(*data: Union[Columnable, AbstractColumn]*)

Convert data to a meerkat column using the appropriate Column type.

full_length()

classmethod get_writer(*mmap: bool = False*, *template: Optional[AbstractColumn] = None*)

head(*n: int = 5*) → *AbstractColumn*

Get the first *n* examples of the column.

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

sample(*n: Optional[int] = None*, *frac: Optional[float] = None*, *replace: bool = False*, *weights: Optional[Union[str, ndarray]] = None*, *random_state: Optional[Union[int, RandomState]] = None*) → *AbstractColumn*

Select a random sample of rows from Column. Roughly equivalent to `sample` in Pandas https://pandas.pydata.org/docs/reference/api/panda s.DataFrame.sample.html.

Parameters

- **n** (*int*) – Number of samples to draw. If *frac* is specified, this parameter should not be passed. Defaults to 1 if *frac* is not passed.
- **frac** (*float*) – Fraction of rows to sample. If *n* is specified, this parameter should not be passed.
- **replace** (*bool*) – Sample with or without replacement. Defaults to False.
- **weights** (*np.ndarray*) – Weights to use for sampling. If *None* (default), the rows will be sampled uniformly. If a numpy array, the sample will be weighted accordingly. If weights do not sum to 1 they will be normalized to sum to 1.
- **random_state** (*Union[int, np.random.RandomState]*) – Random state or seed to use for sampling.

Returns
A random sample of rows from the DataPanel.

Return type
AbstractColumn

sort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *AbstractColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

AbstractColumn

streamlit()

tail(*n: int = 5*) → *AbstractColumn*

Get the last *n* examples of the column.

to_pandas() → Series

Columnable

alias of Union[Sequence, ndarray, Series, Tensor]

property data

Get the underlying data.

property formatter: Callable

property is_mmap

logdir: Path = PosixPath('/home/docs/meerkat')

property metadata

class ArrowArrayColumn(*data: Sequence*, **args*, ***kwargs*)

Bases: *AbstractColumn*

block_class

alias of *ArrowBlock*

classmethod concat(*columns: Sequence[*ArrowArrayColumn*]*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters

other (AbstractColumn) – [description]

to_numpy()

to_pandas()

to_tensor()

class AudioColumn(*data: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*, **args*, ***kwargs*)

Bases: *FileColumn*

A lambda column where each cell represents an audio file on disk. The underlying data is a *PandasSeriesColumn* of strings, where each string is the path to an image. The column materializes the images into memory when indexed. If the column is lazy indexed with the `lz` indexer, the images are not materialized and an `FileCell` or an `AudioColumn` is returned instead.

Parameters

- **data** (*Sequence[str]*) – A list of filepaths to images.
- **transform** (*callable*) – A function that transforms the image (e.g. `torchvision.transforms.functional.center_crop`).

  Warning: In order for the column to be serializable, the transform function must be pickleable.

- **loader** (*callable*) – A callable with signature `def loader(filepath: str) -> PIL.Image:`. Defaults to `torchvision.datasets.folder.default_loader`.

  Warning: In order for the column to be serializable with `write()`, the loader function must be pickleable.

- **base_dir** (*str*) – A base directory that the paths in `data` are relative to. If `None`, the paths are assumed to be absolute.

collate(*batch*)

Collate data.

classmethod default_loader(**args*, ***kwargs*)

class CellColumn(*cells: Optional[Sequence[AbstractCell]] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*

static concat(*columns: Sequence[CellColumn]*)

classmethod from_cells(*cells: Sequence[AbstractCell]*, **args*, ***kwargs*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters

other (*AbstractColumn*) – [description]

property cells

class DataPanel(*data: Optional[Union[dict, list]] = None*, **args*, ***kwargs*)

Bases: *CloneableMixin*, *FunctionInspectorMixin*, *LambdaMixin*, *MappableMixin*, *MaterializationMixin*, *ProvenanceMixin*

Meerkat DataPanel class.

add_column(*name: str*, *data: Union[Sequence, ndarray, Series, Tensor]*, *overwrite=False*) → None

Add a column to the DataPanel.

append(*dp: DataPanel*, *axis: Union[str, int] = 'rows'*, *suffixes: Tuple[str] = None*, *overwrite: bool = False*) → *DataPanel*

Append a batch of data to the dataset.

example_or_batch must have the same columns as the dataset (regardless of what columns are visible).

batch(*batch_size: int = 1*, *drop_last_batch: bool = False*, *num_workers: int = 0*, *materialize: bool = True*, *shuffle: bool = False*, **args*, ***kwargs*)

Batch the dataset. TODO:

Parameters

- **batch_size** – integer batch size
- **drop_last_batch** – drop the last batch if its smaller than batch_size

Returns
batches of data

consolidate()

filter(*function: Optional[Callable] = None*, *with_indices=False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: int = 0*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → Optional[*DataPanel*]

Filter operation on the DataPanel.

classmethod from_arrow(*table: Table*)

Create a Dataset from a pandas DataFrame.

classmethod from_batch(*batch: Dict[str, Union[List,* AbstractColumn*]]*) → *DataPanel*

Convert a batch to a Dataset.

classmethod from_batches(*batches: Sequence[Dict[str, Union[List,* AbstractColumn*]]]*) → *DataPanel*

Convert a list of batches to a dataset.

classmethod from_csv(*filepath: str*, **args*, ***kwargs*)

Create a Dataset from a csv file.

Parameters

- **filepath** (*str*) – The file path or buffer to load from. Same as `pandas.read_csv()`.
- ***args** – Argument list for `pandas.read_csv()`.
- ****kwargs** – Keyword arguments for `pandas.read_csv()`.

Returns
The constructed datapanel.

Return type
DataPanel

classmethod from_dict(*d: Dict*) → *DataPanel*

Convert a dictionary to a dataset.

Alias for Dataset.from_batch(..).

classmethod from_feather(*path: str*)

Create a Dataset from a feather file.

classmethod from_huggingface(**args*, ***kwargs*)

Load a Huggingface dataset as a DataPanel.

Use this to replace *datasets.load_dataset*, so

```
>>> dict_of_datasets = datasets.load_dataset('boolq')
```

becomes

```
>>> dict_of_datapanels = DataPanel.from_huggingface('boolq')
```

classmethod from_jsonl(*json_path: str*) → *DataPanel*

Load a dataset from a .jsonl file on disk, where each line of the json file consists of a single example.

classmethod from_pandas(*df: DataFrame*)

Create a Dataset from a pandas DataFrame.

groupby(**args*, ***kwargs*)

head(*n: int = 5*) → *DataPanel*

Get the first *n* examples of the DataPanel.

items()

keys()

map(*function: Optional[Callable] = None*, *with_indices: bool = False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *drop_last_batch: bool = False*, *num_workers: int = 0*, *output_type: Union[type, Dict[str, type]] = None*, *mmap: bool = False*, *mmap_path: str = None*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → Optional[Union[Dict, List, *AbstractColumn*]]

mean(**args*, ***kwargs*) → *DataPanel*

merge(*right:* DataPanel, *how: str = 'inner'*, *on: Optional[Union[str, List[str]]] = None*, *left_on: Optional[Union[str, List[str]]] = None*, *right_on: Optional[Union[str, List[str]]] = None*, *sort: bool = False*, *suffixes: Sequence[str] = ('_x', '_y')*, *validate=None*)

classmethod read(*path: str*, **args*, ***kwargs*) → *DataPanel*

Load a DataPanel stored on disk.

remove_column(*column: str*) → None

Remove a column from the dataset.

sample(*n: Optional[int] = None*, *frac: Optional[float] = None*, *replace: bool = False*, *weights: Optional[Union[str, ndarray]] = None*, *random_state: Optional[Union[int, RandomState]] = None*) → *DataPanel*

Select a random sample of rows from DataPanel. Roughly equivalent to `sample` in Pandas https://pandas.pydata.org/docs/reference/api/panda s.DataFrame.sample.html.

Parameters

- **n** (*int*) – Number of samples to draw. If *frac* is specified, this parameter should not be passed. Defaults to 1 if *frac* is not passed.
- **frac** (*float*) – Fraction of rows to sample. If *n* is specified, this parameter should not be passed.
- **replace** (*bool*) – Sample with or without replacement. Defaults to False.

- **weights** (*Union[str, np.ndarray]*) – Weights to use for sampling. If *None* (default), the rows will be sampled uniformly. If a numpy array, the sample will be weighted accordingly. If a string, the weights will be applied to the rows based on the column with the name specified. If weights do not sum to 1 they will be normalized to sum to 1.
- **random_state** (*Union[int, np.random.RandomState]*) – Random state or seed to use for sampling.

Returns
A random sample of rows from the DataPanel.

Return type
DataPanel

sort(*by: Union[str, List[str]]*, *ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *DataPanel*

Sort the DataPanel by the values in the specified columns. Similar to `sort_values` in pandas.

Parameters

- **by** (*Union[str, List[str]]*) – The columns to sort by.
- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*.Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A sorted view of DataPanel.

Return type
DataPanel

streamlit()

tail(*n: int = 5*) → *DataPanel*

Get the last *n* examples of the DataPanel.

to_jsonl(*path: str*) → None

Save a Dataset to a jsonl file.

to_pandas() → DataFrame

Convert a Dataset to a pandas DataFrame.

update(*function: Optional[Callable] = None*, *with_indices: bool = False*, *input_columns: Optional[Union[str, List[str]]] = None*, *is_batched_fn: bool = False*, *batch_size: Optional[int] = 1*, *remove_columns: Optional[List[str]] = None*, *num_workers: int = 0*, *output_type: Union[type, Dict[str, type]] = None*, *mmap: bool = False*, *mmap_path: str = None*, *materialize: bool = True*, *pbar: bool = False*, ***kwargs*) → *DataPanel*

Update the columns of the dataset.

values()

write(*path: str*) → None

Save a DataPanel to disk.

property columns

Column names in the DataPanel.

property data: *BlockManager*

Get the underlying data (excluding invisible rows).

To access underlying data with invisible rows, use *_data*.

logdir: Path = PosixPath('/home/docs/meerkat')

property ncols

Number of rows in the DataPanel.

property nrows

Number of rows in the DataPanel.

property shape

Shape of the DataPanel (num_rows, num_columns).

class FileCell(*data: LambdaCellOp*)

Bases: *LambdaCell*

from_filepath(*transform: Optional[callable] = None*, *loader: Optional[callable] = None*, *path: Optional[str] = None*, *base_dir: Optional[str] = None*)

property absolute_path

class FileColumn(*data: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*, **args*, ***kwargs*)

Bases: *LambdaColumn*

A column where each cell represents an file stored on disk or the web. The underlying data is a *PandasSeriesColumn* of strings, where each string is the path to a file. The column materializes the files into memory when indexed. If the column is lazy indexed with the `lz` indexer, the files are not materialized and a `FileCell` or a `FileColumn` is returned instead.

Parameters

- **data** (*Sequence[str]*) – A list of filepaths to images.
- **transform** (*callable*) – A function that transforms the image (e.g. `torchvision.transforms.functional.center_crop`).

  Warning: In order for the column to be serializable, the transform function must be pickleable.

- **loader** (*callable*) – A callable with signature `def loader(filepath: str) -> PIL.Image:`. Defaults to `torchvision.datasets.folder.default_loader`.

  Warning: In order for the column to be serializable with `write()`, the loader function must be pickleable.

- **base_dir** (*str*) – A base directory that the paths in `data` are relative to. If `None`, the paths are assumed to be absolute.

classmethod default_loader(**args*, ***kwargs*)

classmethod from_filepaths(*filepaths: Sequence[str]*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

property base_dir

property loader

property transform

class ImageColumn(*data: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, *base_dir: Optional[str] = None*, **args*, ***kwargs*)

Bases: *FileColumn*

A column where each cell represents an image stored on disk. The underlying data is a *PandasSeriesColumn* of strings, where each string is the path to an image. The column materializes the images into memory when indexed. If the column is lazy indexed with the `lz` indexer, the images are not materialized and an `ImageCell` or an `ImageColumn` is returned instead.

Parameters

- **data** (*Sequence[str]*) – A list of filepaths to images.
- **transform** (*callable*) – A function that transforms the image (e.g. `torchvision.transforms.functional.center_crop`).

  Warning: In order for the column to be serializable, the transform function must be pickleable.

- **loader** (*callable*) – A callable with signature `def loader(filepath: str) -> PIL.Image:`. Defaults to `torchvision.datasets.folder.default_loader`.

  Warning: In order for the column to be serializable with `write()`, the loader function must be pickleable.

- **base_dir** (*str*) – A base directory that the paths in `data` are relative to. If `None`, the paths are assumed to be absolute.

classmethod default_loader(**args*, ***kwargs*)

class LambdaCell(*data: LambdaCellOp*)

Bases: *AbstractCell*

get(**args*, ***kwargs*)

Get me the thing that this cell exists for.

property data: object

Get the data associated with this cell.

class LambdaColumn(*data: Union[LambdaOp, BlockView]*, *output_type: Optional[type] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*

block_class

alias of LambdaBlock

static concat(*columns: Sequence[LambdaColumn]*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (*AbstractColumn*) – [description]

property fn: Callable

Subclasses like *ImageColumn* should be able to implement their own version.

class ListColumn(*data: Optional[Sequence] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*

batch(*batch_size: int = 1*, *drop_last_batch: bool = False*, *collate: bool = True*, **args*, ***kwargs*)

Batch the column.

Parameters

- **batch_size** – integer batch size
- **drop_last_batch** – drop the last batch if its smaller than batch_size
- **collate** – whether to collate the returned batches

Returns
batches of data

classmethod concat(*columns: Sequence[ListColumn]*)

default_formatter()

classmethod from_list(*data: Sequence*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (*AbstractColumn*) – [description]

class MedicalVolumeCell(*paths: Union[str, Path, PathLike, Sequence[Union[str, Path, PathLike]]]*, *loader: Optional[Callable] = None*, *transform: Optional[Callable] = None*, *cache_metadata: bool = False*, **args*, ***kwargs*)

Bases: *PathsMixin*, *AbstractCell*

Interface for loading medical volume data.

Examples

Specify xray dicoms with default orientation (`"SI", "AP"`): >>> cell = MedicalVolumeCell("/path/to/xray.dcm", loader=DicomReader(group_by=None, default_ornt=("SI", "AP"))

Load multi-echo MRI volumes >>> cell = MedicalVolumeCell("/path/to/mri/scan/dir", loader=DicomReader(group_by="EchoNumbers"))

clear_metadata()

classmethod default_loader(*paths: Sequence[Path]*, **args*, ***kwargs*)

classmethod from_state(*state*, **args*, ***kwargs*)

get(**args*, *cache_metadata: Optional[bool] = None*, ***kwargs*)

Get me the thing that this cell exists for.

get_metadata(*ignore_bytes: bool = False*, *readable: bool = False*, *as_raw_type: bool = False*, *force_load: bool = False*) → Dict

get_state()

class MedicalVolumeColumn(**args*, ***kwargs*)

Bases: *CellColumn*

classmethod from_filepaths(*filepaths: Optional[Sequence[str]] = None*, *loader: Optional[callable] = None*, *transform: Optional[callable] = None*, **args*, ***kwargs*)

class NumpyArrayColumn(*data: Sequence*, **args*, ***kwargs*)

Bases: *AbstractColumn*, `NDArrayOperatorsMixin`

block_class

alias of *NumpyBlock*

argsort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *NumpyArrayColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
NumpySeriesColumn

For now! Raises error when shape of input array is more than one error.

classmethod concat(*columns: Sequence[NumpyArrayColumn]*)

classmethod from_array(*data: ndarray*, **args*, ***kwargs*)

classmethod from_npy(*path*, *mmap_mode=None*, *allow_pickle=False*, *fix_imports=True*, *encoding='ASCII'*)

classmethod get_writer(*mmap: bool = False*, *template: Optional[AbstractColumn] = None*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

sort(*ascending: Union[bool, List[bool]] = True*, *axis: int = -1*, *kind: str = 'quicksort'*, *order: Optional[Union[str, List[str]]] = None*) → *NumpyArrayColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
AbstractColumn

to_numpy() → ndarray

to_pandas() → Series

to_tensor() → Tensor

Use *column.to_tensor()* instead of *torch.tensor(column)*, which is very slow.

property is_mmap

class PandasSeriesColumn(*data: Optional[Sequence] = None*, *collate_fn: Optional[Callable] = None*, *formatter: Optional[Callable] = None*, **args*, ***kwargs*)

Bases: *AbstractColumn*, NDArrayOperatorsMixin

block_class

alias of *PandasBlock*

cat

alias of _MeerkatCategoricalAccessor

dt

alias of _MeerkatCombinedDatetimelikeProperties

str

alias of _MeerkatStringMethods

argsort(*ascending: bool = True*, *kind: str = 'quicksort'*) → *PandasSeriesColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type

PandasSeriesColumn

For now! Raises error when shape of input array is more than one error.

classmethod concat(*columns: Sequence[PandasSeriesColumn]*)

classmethod from_array(*data: ndarray*, **args*, ***kwargs*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters
other (AbstractColumn) – [description]

sort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *PandasSeriesColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A view of the column with the sorted data.

Return type
AbstractColumn

to_numpy() → Tensor

to_pandas() → Series

to_tensor() → Tensor

Use *column.to_tensor()* instead of *torch.tensor(column)*, which is very slow.

class SpacyColumn(*data: Sequence[spacy_tokens.Doc] = None*, **args*, ***kwargs*)

Bases: ListColumn

classmethod from_docs(*data: Sequence[spacy_tokens.Doc]*, **args*, ***kwargs*)

classmethod from_texts(*texts: Sequence[str]*, *lang: str = 'en_core_web_sm'*, **args*, ***kwargs*)

classmethod read(*path: str*, *nlp: spacy.language.Language = None*, *lang: str = None*, **args*, ***kwargs*) → *SpacyColumn*

write(*path: str*, ***kwargs*) → None

property docs

property tokens

class TensorColumn(*data: Optional[Sequence] = None*, **args*, ***kwargs*)

Bases: NDArrayOperatorsMixin, AbstractColumn

block_class

alias of *TensorBlock*

argsort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *TensorColumn*

Return indices that would sorted the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

TensorColumn

For now! Raises error when shape of input array is more than one error.

classmethod concat(*columns: Sequence[*TensorColumn*]*)

classmethod from_data(*data: Union[Sequence, ndarray, Series, Tensor,* AbstractColumn*]*)

Convert data to an EmbeddingColumn.

classmethod get_writer(*mmap: bool = False*, *template: Optional[*AbstractColumn*] = None*)

is_equal(*other:* AbstractColumn) → bool

Tests whether two columns.

Parameters

other (AbstractColumn) – [description]

sort(*ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *TensorColumn*

Return a sorted view of the column.

Parameters

- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*. Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns

A view of the column with the sorted data.

Return type

AbstractColumn

to_numpy() → Series

to_pandas() → Series

to_tensor() → Tensor

class provenance(*enabled: bool = True*)

Bases: object

concat(*objs: Union[Sequence[DataPanel], Sequence[AbstractColumn]]*, *axis: Union[str, int] = 'rows'*, *suffixes: Tuple[str] = None*, *overwrite: bool = False*) → Union[*DataPanel*, *AbstractColumn*]

Concatenate a sequence of columns or a sequence of *DataPanel`s. If sequence is empty, returns an empty `DataPanel*.

- If concatenating columns, all columns must be of the same type (e.g. all

ListColumn). - If concatenating `DataPanel`s along axis 0 (rows), all `DataPanel`s must have the same set of columns. - If concatenating `DataPanel`s along axis 1 (columns), all `DataPanel`s must have the same length and cannot have any of the same column names.

Parameters

- **objs** (*Union[Sequence[DataPanel], Sequence[AbstractColumn]]*) – sequence of columns or DataPanels.
- **axis** (*Union[str, int]*) – The axis along which to concatenate. Ignored if concatenating columns.

Returns

concatenated DataPanel or column

Return type

Union[*DataPanel*, *AbstractColumn*]

embed(*data: DataPanel*, *input: str*, *encoder: Union[str, Encoder] = 'clip'*, *modality: Optional[str] = None*, *out_col: Optional[str] = None*, *device: Union[int, str] = 'cpu'*, *mmap_dir: Optional[str] = None*, *num_workers: int = 4*, *batch_size: int = 128*, ***kwargs*) → *DataPanel*

Embed a column of data with an encoder from the encoder registry.

Examples

Suppose you have an Image dataset (e.g. Imagenette, CIFAR-10) loaded into a Meerkat DataPanel. You can embed the images in the dataset with CLIP using a code snippet like:

```
import meerkat as mk

dp = mk.datasets.get("imagenette")

dp = mk.embed(
    data=dp,
    input_col="img",
    encoder="clip"
)
```

Parameters

- **data** (*mk.DataPanel*) – A DataPanel containing the data to embed.
- **input_col** (*str*) – The name of the column to embed.
- **encoder** (*Union[str, Encoder], optional*) – Name of the encoder to use. List supported encoders with `domino.encoders`. Defaults to "clip". Alternatively, pass an `Encoder` object containing a custom encoder.
- **modality** (*str, optional*) – The modality of the data to be embedded. Defaults to None, in which case the modality is inferred from the type of the input column.

- **out_col** (*str, optional*) – The name of the column where the embeddings are stored. Defaults to None, in which case it is `"{encoder}({input_col})"`.
- **device** (*Union[int, str], optional*) – The device on which. Defaults to "cpu".
- **mmap_dir** (*str, optional*) – The path to directory where a memory-mapped file containing the embeddings will be written. Defaults to None, in which case the embeddings are not memmapped.
- **num_workers** (*int, optional*) – Number of worker processes used to load the data from disk. Defaults to 4.
- **batch_size** (*int, optional*) – Size of the batches to used . Defaults to 128.
- ****kwargs** – Additional keyword arguments are passed to the encoder. To see supported arguments for each encoder, see the encoder documentation (e.g. `clip()`).

Returns
A view of `data` with a new column containing the embeddings. This column will be named according to the `out_col` parameter.

Return type
mk.DataPanel

get(*name: str*, *dataset_dir: Optional[str] = None*, *version: Optional[str] = None*, *download_mode: str = 'reuse'*, *registry: Optional[str] = None*, ***kwargs*) → Union[*DataPanel*, Dict[str, *DataPanel*]]

Load a dataset into .

Parameters

- **name** (*str*) – Name of the dataset.
- **dataset_dir** (*str*) – The directory containing dataset data. Defaults to *~/.meerkat/datasets/{name}*.
- **version** (*str*) – The version of the dataset. Defaults to *latest*.
- **download_mode** (*str*) – The download mode. Options are: "reuse" (default) will download the dataset if it does not exist, "force" will download the dataset even if it exists, "extract" will reuse any downloaded archives but force extracting those archives, and "skip" will not download the dataset if it doesn't yet exist. Defaults to *reuse*.
- ****kwargs** – Additional arguments passed to the dataset.

merge(*left: DataPanel*, *right: DataPanel*, *how: str = 'inner'*, *on: Union[str, List[str]] = None*, *left_on: Union[str, List[str]] = None*, *right_on: Union[str, List[str]] = None*, *sort: bool = False*, *suffixes: Sequence[str] = ('_x', '_y')*, *validate=None*)

sample(*data: Union[DataPanel, AbstractColumn]*, *n: Optional[int] = None*, *frac: Optional[float] = None*, *replace: bool = False*, *weights: Optional[Union[str, ndarray]] = None*, *random_state: Optional[Union[int, RandomState]] = None*) → Union[*DataPanel*, *AbstractColumn*]

Select a random sample of rows from DataPanel or Column. Roughly equivalent to `sample` in Pandas https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html.

Parameters

- **data** (*Union[DataPanel, AbstractColumn]*) – DataPanel or Column to sample from.
- **n** (*int*) – Number of samples to draw. If *frac* is specified, this parameter should not be passed. Defaults to 1 if *frac* is not passed.
- **frac** (*float*) – Fraction of rows to sample. If *n* is specified, this parameter should not be passed.

- **replace** (*bool*) – Sample with or without replacement. Defaults to False.
- **weights** (*Union[str, np.ndarray]*) – Weights to use for sampling. If *None* (default), the rows will be sampled uniformly. If a numpy array, the sample will be weighted accordingly. If a string and *data* is a DataPanel, the weights will be applied to the rows based on the column with the name specified. If weights do not sum to 1 they will be normalized to sum to 1.
- **random_state** (*Union[int, np.random.RandomState]*) – Random state or seed to use for sampling.

Returns

A random sample of rows from DataPanel or
Column.

Return type
Union[*DataPanel*, *AbstractColumn*]

sort(*data:* DataPanel, *by: Union[str, List[str]]*, *ascending: Union[bool, List[bool]] = True*, *kind: str = 'quicksort'*) → *DataPanel*

Sort a DataPanel or Column. If a DataPanel, sort by the values in the specified columns. Similar to `sort_values` in pandas.

Parameters

- **data** (*Union[DataPanel, AbstractColumn]*) – DataPanel or Column to sort.
- **by** (*Union[str, List[str]]*) – The columns to sort by. Ignored if data is a Column.
- **ascending** (*Union[bool, List[bool]]*) – Whether to sort in ascending or descending order. If a list, must be the same length as *by*.Defaults to True.
- **kind** (*str*) – The kind of sort to use. Defaults to 'quicksort'. Options include 'quicksort', 'mergesort', 'heapsort', 'stable'.

Returns
A sorted view of DataPanel.

Return type
DataPanel

PYTHON MODULE INDEX

m

INDEX

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W