colabfit-tools

Release 0.1

ColabFit

Nov 23, 2021

CONTENTS:

`colabfit-tools` is a package for constructing, manipulating, and exploring datasets for data-driven interatomic potentials. This package is part of the ColabFit project at the University of Minnesota.

Despite the rapidly increasing popularity of data-driven interatomic potentials (DDIPs), training data for these models has historically been difficult to obtain and work with. Data is often stored in internally-developed formats, insufficiently documented, and not made readily available to the public.

`colabfit_tools` aims to address these problems by defining a standard method for representing DDIP training datasets programatically, and providing software tools for manipulating these representations.

CHAPTER

ONE

OVERVIEW

1.1 Dataset Standard

The *Dataset* class was designed to be able to be as flexible as possible to incorporate different types of data (computational/experimental) while also making the datasets efficient to query, store, and manipulate. A *Dataset* is constructed using five core data structures:

- ***Configuration* (CO):** The information necessary to uniquely define the input to a calculation (e.g., atomic types, positions, lattice vectors, constraints, ...)
- ***Property* (PR):** The outputs from a calculation (e.g., DFT-computed energies/forces/stresses)
- ***PropertySettings* (PSO):** Additional metadata that useful for setting up the calculation (e.g., software version, input files, ...)
- ***ConfigurationSet* (CS):** An object defining a group of *Configuration* instances and providing useful metadata for organizing datasets (e.g., "Snapshots from a molecular dynamics run at 1000K")
- ***Dataset* (DS):** The top-level dataset, which aggregates information up from the configuration sets and subdatasets for improved discoverability

Fig. 1: A diagram showing the relationship between the five core data structures

CHAPTER

TWO

GETTING STARTED

2.1 Installation

Currently, installation is only supported using `pip` to install directly from the private GitHub repository.

2.1.1 Using pip

```
$ pip install git+https://<username_or_pat>@github.com/colabfit/colabfit-tools.git
```

Note that since `colabfit-tools` is currently still a private project, `<username_or_pat>` must either be your GitHub username (if you have access to the repository) or a Personal Access Token that has appropriate permissions.

CHAPTER

THREE

EXAMPLES

3.1 Basic example

This example corresponds to the `colabfit/examples/basic_example.ipynb` Jupyter notebook in the GitHub repo, which can be run in Google Colab.

3.1.1 Creating a Dataset from scratch

Initialize the Dataset and add basic metadata.

```
from colabfit.tools.dataset import Dataset

dataset = Dataset(name='example')

dataset.authors = [
    'J. E. Lennard-Jones',
]

dataset.links = [
    'https://en.wikipedia.org/wiki/John_Lennard-Jones'
]

dataset.description = "This is an example dataset"
```

3.1.2 Adding Configurations

Load the configurations onto the Dataset by either manually constructing the Configurations and assigning them to the `configurations` attribute, or using *`load_data()`*, which calls a pre-made Converter and returns a list of Configuration objects.

Manually

```
import numpy as np
from ase import Atoms

images = []
for i in range(1, 1000):
    atoms = Atoms('H'*i, positions=np.random.random((i, 3)))

    atoms.info['_name'] = 'configuration_' + str(i)

    atoms.info['dft_energy'] = i*i
    atoms.arrays['dft_forces'] = np.random.normal(size=(i, 3))

    images.append(atoms)
```

```
from colabfit.tools.configuration import Configuration

dataset.configurations = [
    Configuration.from_ase(atoms) for atoms in images
]
```

Using `load_data()`

Note that the file_format must be specified, and a name_field (or None) should be provided to specify the name of the loaded configurations.

```
from ase.io import write

write('/content/example.extxyz', images)
```

```
from colabfit.tools.dataset import load_data

dataset.configurations = load_data(
        file_path='/content/example.extxyz',
        file_format='xyz',
        name_field='_name',
        elements=['H'],
        default_name=None,
)
```

3.1.3 Applying labels to configurations

Labels can be specified as lists or single strings (which will be wrapped in a list) Metadata can be applied to individual configurations using labels. Labels are applied by matching a regular expression to `configuration.info[ASE_NAME_FIELD]` for each configuration. Regex mappings are provided by setting the configuration_label_regexes dictionary.

```
dataset.configuration_label_regexes = {
    'configuration_[1-10]': 'small',
```

(continues on next page)

(continued from previous page)

```
    '.*': 'random',
}
```

3.1.4 Configuration sets using regexes

A *ConfigurationSet* can be used to create groups of configurations for organizational purposes. This can be done in a similar manner to how configuration labels are applied, but using the `configuration_set_regexes` dictionary. Note that a configuration may exist in multiple sets at the same time.

```
dataset.configuration_set_regexes = {
    'configuration_[1-499]':   "The first configuration set",
    'configuration_[500-999]': "The second configuration set",
}
```

3.1.5 Synchronizing the dataset

A Dataset is a pool of configurations and properties, where the configurations are further organized by grouping them into configuration sets, and the properties are linked to property settings. A Dataset then aggregates information up from the configurations, properties, and property settings. In order to ensure that the information applied by specifying `configuration_label_regexes`, `configuration_set_regexes`, and `property_settings_regexes` are up-to-date, `dataset.resync()` should be called before performing critical operations like saving a Dataset. Some core functions will call `resync()` automatically.

```
dataset.resync()
```

3.1.6 Parsing the data

Parse the properties by specifying a `property_map`, which is a special dictionary on a Dataset. See *Property map* for more details.

```
dataset.property_map = {
    'default': {
        'energy': {'field': 'dft_energy', 'units': 'eV'},
        'forces': {'field': 'dft_forces', 'units': 'eV/Ang'},
    }
}
```

```
dataset.parse_data(convert_units=False, verbose=True)
```

3.1.7 Exploring the data

Use *get_data()* to obtain a list of the given property field where each element has been wrapped in a numpy array.

```
energies = dataset.get_data('energy', ravel=True)
forces   = dataset.get_data('forces', ravel=True)
```

Basic statistics can be obtained using *get_statistics()*.

```
# Returns: {'average': ..., 'std': ..., 'min': ..., 'max':, ..., 'average_abs': ...}
dataset.get_statistics('energy')
```

The *plot_histograms()* function is useful for quickly visualizing the distribution of your data.

```
dataset.plot_histograms(['energy', 'forces'])
```

3.1.8 Providing calculation metadata

Metadata for computing properties can be provided by constructing a *PropertySettings* object and matching it to a property by regex matching on the property's linked configurations. It is good practice to always attach a PropertySettings object to every Property to improve reproducibility of the data. It is especially useful to include example input files using the `files` field.

```
from colabfit.tools.property_settings import PropertySettings

dataset.property_settings_regexes = {
    '.*':
        PropertySettings(
            method='VASP',
            description='energy/force calculations',
            files=['/path/to/INCAR'],
            labels=['PBE', 'GGA'],
        )
}
```

3.2 QM9 example

This example will be used to highlight some of the more advanced features of the Dataset class using the popular QM9 dataset. It is suggested that you go through the *basic example* first. The complete code will not be shown in this example (for the complete code, see the Jupyter notebook at `colabfit/examples/qm9.ipynb`); instead, only the additional features will be discussed here.

Note that this example assumes that the raw data has already been downloaded using the following commands:

```
$ mkdir qm9
$ cd qm9 && wget -O dsgdb9nsd.xyz.tar.bz2 https://figshare.com/ndownloader/files/3195389␣
↪&& tar -xvjf dsgdb9nsd.xyz.tar.bz2
```

3.2.1 Writing a custom property for QM9

The QM9 dataset contains a large number of computed properties for each Configuration, as documented in its original README.

In order to preserve this information, a custom property can be defined (see *Custom properties* for more information).

```
qm9_property_definition = {
    'property-id': 'qm9-property',
    'property-title': 'A, B, C, mu, alpha, homo, lumo, gap, r2, zpve, U0, U, H, G, Cv',
    'property-description': 'Geometries minimal in energy, corresponding harmonic␣
↪frequencies, dipole moments, polarizabilities, along with energies, enthalpies, and␣
↪free energies of atomization',
    'a':     {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Rotational constant A'},
    'b':     {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Rotational constant B'},
    'c':     {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Rotational constant C'},
    'mu':    {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Dipole moment'},
    'alpha': {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Isotropic polarizability'},
    'homo':  {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Energy of Highest occupied molecular orbital (HOMO)'},
    'lumo':  {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Energy of Lowest occupied molecular orbital (LUMO)'},
    'gap':   {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Gap, difference between LUMO and HOMO'},
    'r2':    {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Electronic spatial extent'},
    'zpve':  {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Zero point vibrational energy'},
    'u0':    {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Internal energy at 0 K'},
    'u':     {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Internal energy at 298.15 K'},
    'h':     {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Enthalpy at 298.15 K'},
    'g':     {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Free energy at 298.15 K'},
    'cv':    {'type': 'float', 'has-unit': True, 'extent': [], 'required': True,
↪'description': 'Heat capacity at 298.15 K'},
    'smiles-relaxed':    {'type': 'string', 'has-unit': False, 'extent': [], 'required':␣
↪True, 'description': 'SMILES for relaxed geometry'},
    'inchi-relaxed':     {'type': 'string', 'has-unit': False, 'extent': [], 'required':␣
↪True, 'description': 'InChI for relaxed geometry'},
}
```

```
dataset.custom_definitions = {'qm9-property': qm9_property_definition}
```

Note that a property definition is used for performing verification checks when parsing the data. A `property_map` must still be provided for specifying _how_ to parse the data and what the units of the fields are.

```
property_map = {
    'qm9-property': {
        # Property Definition field: {'field': ASE field, 'units': ASE-readable units}
        'a':     {'field': 'A',     'units': 'GHz'},
        'b':     {'field': 'B',     'units': 'GHz'},
        'c':     {'field': 'C',     'units': 'GHz'},
        'mu':    {'field': 'mu',    'units': 'Debye'},
        'alpha': {'field': 'alpha', 'units': 'Bohr*Bohr*Bohr'},
        'homo':  {'field': 'homo',  'units': 'Hartree'},
        'lumo':  {'field': 'lumo',  'units': 'Hartree'},
        'gap':   {'field': 'gap',   'units': 'Hartree'},
        'r2':    {'field': 'r2',    'units': 'Bohr*Bohr'},
        'zpve':  {'field': 'zpve',  'units': 'Hartree'},
        'u0':    {'field': 'U0',    'units': 'Hartree'},
        'u':     {'field': 'U',     'units': 'Hartree'},
        'h':     {'field': 'H',     'units': 'Hartree'},
        'g':     {'field': 'G',     'units': 'Hartree'},
        'cv':    {'field': 'Cv',    'units': 'cal/mol/K'},
        'smiles-relaxed': {'field': 'SMILES_relaxed', 'units': None},
        'inchi-relaxed': {'field': 'SMILES_relaxed',  'units': None},
    }
}
```

```
dataset.property_map = property_map
```

3.2.2 Defining a reader function

Since the data in QM9 is not stored in a typical format (it uses an uncommon modification to the typical XYZ format), it is necessary to use the *FolderConverter* class, with a custom `reader()` function.

```
def reader(file_path):
    # A function for returning a list of ASE a

    properties_order = [
        'tag', 'index', 'A', 'B', 'C', 'mu', 'alpha', 'homo', 'lumo', 'gap', 'r2', 'zpve
↪', 'U0', 'U', 'H', 'G', 'Cv'
    ]

    images = []
    with open(file_path, 'r') as f:
        lines = [_.strip() for _ in f.readlines()]

        na = int(lines[0])
        properties = lines[1].split()

        symbols = []
        positions = []
        partial_charges = []

        for line in lines[2:2+na]:
            split = line.split()
```

(continues on next page)

(continued from previous page)

```
            split = [_.replace('*^', 'e') for _ in split]  # Python-readable scientific␣
↪notation

            # Line order: symbol, x, y, z, charge
            symbols.append(split[0])
            positions.append(split[1:4])
            partial_charges.append(split[-1])

        positions = np.array(positions)
        partial_charges = np.array(partial_charges, dtype=float)

        atoms = Atoms(symbols=symbols, positions=positions)

        atoms.info['mulliken_partial_charges'] = partial_charges

        name = os.path.splitext(os.path.split(file_path)[-1])[0]

        atoms.info['name'] = name

        for pname, val in zip(properties_order[2:], properties[2:]):
            atoms.info[pname] = float(val)

        frequencies = np.array(lines[-3].split(), dtype=float)
        atoms.info['frequencies'] = frequencies

        smiles = lines[-2].split()
        inchi  = lines[-1].split()

        atoms.info['SMILES']    = smiles[0]
        atoms.info['SMILES_relaxed'] = smiles[1]
        atoms.info['InChI']     = inchi[0]
        atoms.info['InChI_relaxed']  = inchi[1]

        images.append(atoms)

    return images
```

```
dataset.configurations = load_data(
    file_path='qm9',
    file_format='folder',
    name_field='name',  # key in Configuration.info to use as the Configuration name
    elements=['H', 'C', 'N', 'O', 'F'],    # order matters for CFG files, but not others
    default_name='qm9',  # default name with `name_field` not found
    reader=reader,
    glob_string='*.xyz',
    verbose=True
)
```

3.2.3 Writing to Markdown

To avoid having to re-process the raw data using `reader`, and to provide a cleaner storage format, *`to_markdown()`* can be used (see *Reading/writing Datasets with Markdown* for more details).

```
dataset.to_markdown(
    base_folder='/content',
    html_file_name='README.md',
    data_file_name=dataset.name+'.extxyz',
    data_format='xyz',
)
```

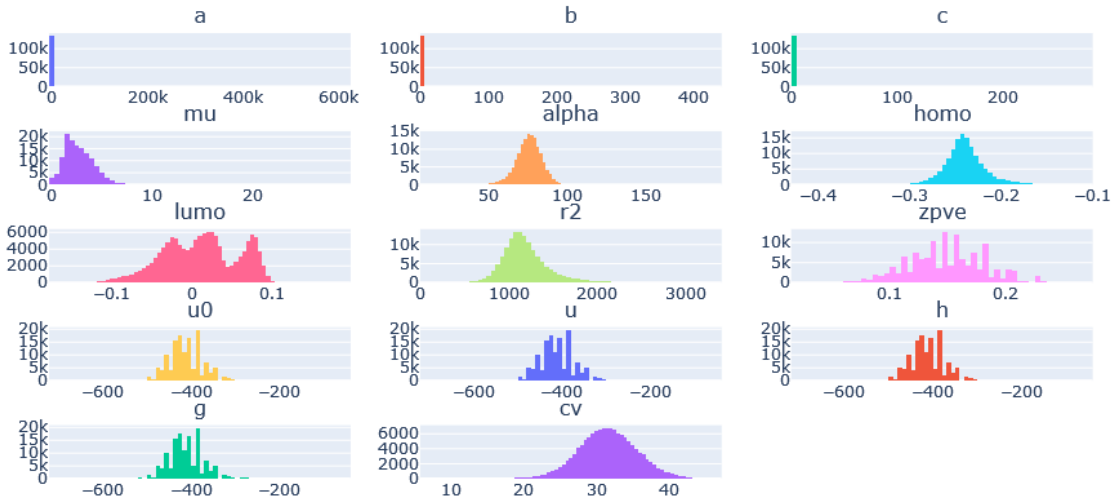
3.2.4 Cleaning the dataset

Using the *`plot_histograms()`* and *`get_statistics()`* functions reveals that the QM9 dataset has some outlying data entries.

```
print(dataset.get_statistics('a'))
print(dataset.get_statistics('b'))
print(dataset.get_statistics('c'))

# {'average': 9.814382088508797, 'std': 1809.4589082320583, 'min': 0.0, 'max': 619867.68314,
↪'average_abs': 9.814382088508797}
# {'average': 1.4060972645920002, 'std': 1.5837889998648804, 'min': 0.33712, 'max': 437.90386,
↪ 'average_abs': 1.4060972645920002}
# {'average': 1.1249210272988013, 'std': 1.0956136904779634, 'min': 0.33118, 'max': 282.94545,
↪ 'average_abs': 1.1249210272988013}
```

```
dataset.plot_histograms([
    'a', 'b', 'c', 'mu', 'alpha', 'homo', 'lumo', 'r2', 'zpve', 'u0', 'u',
    'h', 'g', 'cv'
])
```



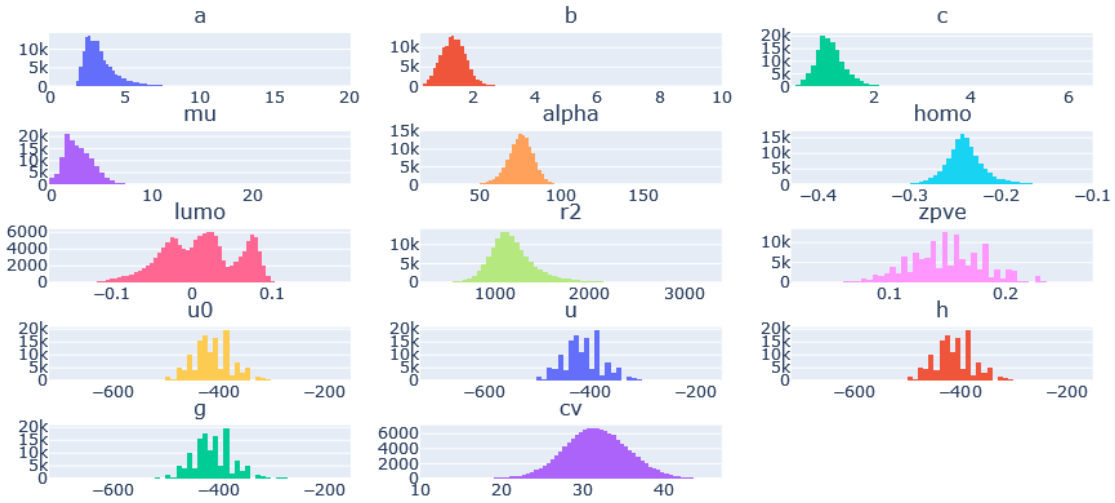
The *filter()* function can be used to return a new dataset without the outlying data.

```
clean = dataset.filter(
    'data',
    lambda p: (p['a']['source-value'] < 20) and (p['b']['source-value'] < 10),
    verbose=True
)
```

```
clean.plot_histograms([
    'a', 'b', 'c', 'mu', 'alpha', 'homo', 'lumo', 'r2', 'zpve', 'u0', 'u',
    'h', 'g', 'cv'
])
```

```
print(clean.get_statistics('a'))
print(clean.get_statistics('b'))
print(clean.get_statistics('c'))

# {'average': 3.407053427070018, 'std': 1.3368223663235594, 'min': 0.0, 'max': 19.99697,
↪'average_abs': 3.407053427070018}
# {'average': 1.3966863945821093, 'std': 0.45813797072575396, 'min': 0.33712, 'max': 9.93509,
↪'average_abs': 1.3966863945821093}
# {'average': 1.1177706236464617, 'std': 0.328798457356026, 'min': 0.33118, 'max': 6.46247,
↪'average_abs': 1.1177706236464617}
```



3.3 Si PRX GAP

This example will be used to highlight some of the more advanced features of the Dataset class using the popular Si GAP dataset. It is suggested that you go through the *basic example* first. The complete code will not be shown in this example (for the complete code, see the Jupyter notebook at `colabfit/examples/si_prx_gap.ipynb`); instead, only the additional features will be discussed here.

Note that this example assumes that the raw data has already been downloaded using the following commands:

```
$ mkdir si_prx_gap
$ cd si_prx_gap && wget -O Si_PRX_GAP.zip https://www.repository.cam.ac.uk/bitstream/
↪handle/1810/317974/Si_PRX_GAP.zip?sequence=1&isAllowed=yield
$ cd si_prx_gap && unzip Si_PRX_GAP.zip
```

3.3.1 Loading from a file

This example uses *load_data()* to load the data from an existing Extended XYZ file. Note that the raw data includes the `config_type` field, which is used to generate the names of the loaded Configurations. A `default_name` is also provided to handle the configurations that do not have a `config_type` field. `verbose=True` is used here since the dataset is large enough to warrant a progress bar.

```
dataset.configurations = load_data(
        file_path='./si_prx_gap/gp_iter6_sparse9k.xml.xyz',
        file_format='xyz',
        name_field='config_type',  # key in Configuration.info to use as the␣
↪Configuration name
        elements=['Si'],
        default_name=dataset.name,  # default name with `name_field` not found
```

(continues on next page)

(continued from previous page)

```
        verbose=True
)
```

3.3.2 Manually constructed ConfigurationSets

Since this dataset was manually constructed by its authors, a large amount of additional information has been provided to better identify the Configurations (see Table I. in the original paper). In order to retain this information, we define ConfigurationSets by regex matching on the Configuration names (see *Building configuration sets* for more details).

```
dataset.configuration_set_regexes = {
    'isolated_atom': 'Reference atom',
    'bt': 'Beta-tin',
    'dia': 'Diamond',
    'sh': 'Simple hexagonal',
    'hex_diamond': 'Hexagonal diamond',
    'bcc': 'Body-centered-cubic',
    'bc8': 'BC8',
    'fcc': 'Face-centered-cubic',
    'hcp': 'Hexagonal-close-packed',
    'st12': 'ST12',
    'liq': 'Liquid',
    'amorph': 'Amorphous',
    'surface_001': 'Diamond surface (001)',
    'surface_110': 'Diamond surface (110)',
    'surface_111': 'Diamond surface (111)',
    'surface_111_pandey': 'Pandey reconstruction of diamond (111) surface',
    'surface_111_3x3_das': 'Dimer-adatom-stacking-fault (DAS) reconstruction',
    '111adatom': 'Configurations with adatom on (111) surface',
    'crack_110_1-10': 'Small (110) crack tip',
    'crack_111_1-10': 'Small (111) crack tip',
    'decohesion': 'Decohesion of diamond-structure Si along various directions',
    'divacancy': 'Diamond divacancy configurations',
    'interstitial': 'Diamond interstitial configurations',
    'screw_disloc': 'Si screw dislocation core',
    'sp': 'sp bonded configurations',
    'sp2': 'sp2 bonded configurations',
    'vacancy': 'Diamond vacancy configurations'
}
```

3.3.3 Manually applied Configuration labels

Similarly, this additional knowledge about the types of Configurations in the dataset can be used to apply metadata labels to the Configurations, which is useful for enabling querying over the data by future users. See *Applying Configuration labels*

```
dataset.configuration_label_regexes = {
    'isolated_atom': 'isolated_atom',
    'bt': 'a5',
    'dia': 'diamond',
    'sh': 'sh',
```

(continues on next page)

(continued from previous page)

```
    'hex_diamond': 'sonsdaleite',
    'bcc': 'bcc',
    'bc8': 'bc8',
    'fcc': 'fcc',
    'hcp': 'hcp',
    'st12': 'st12',
    'liq': 'liquid',
    'amorph': 'amorphous',
    'surface_001': ['surface', '001'],
    'surface_110': ['surface', '110'],
    'surface_111': ['surface', '111'],
    'surface_111_pandey': ['surface', '111'],
    'surface_111_3x3_das': ['surface', '111', 'das'],
    '111adatom': ['surface', '111', 'adatom'],
    'crack_110_1-10': ['crack', '110'],
    'crack_111_1-10': ['crac', '111'],
    'decohesion': ['diamond', 'decohesion'],
    'divacancy': ['diamond', 'vacancy', 'divacancy'],
    'interstitial': ['diamond', 'interstitial'],
    'screw_disloc': ['screw', 'dislocation'],
    'sp': 'sp',
    'sp2': 'sp2',
    'vacancy': ['diamond', 'vacancy']
}
```

3.3.4 Renaming Configuration fields

In order to ensure that *parse_data()* is able to properly parse the data, the fields in `Configuration.info` and `Configuration.arrays` should match those used in `Dataset.property_map`. In the case of the Si GAP dataset, some of the data has incorrectly labeled energy/force/virial fields, where it uses "DFT_*" instead of "dft_*" (lowercase) like the rest of the Configurations. In order to fix this, we use the *rename_configuration_field()* function:

```
dataset.rename_configuration_field('DFT_energy', 'dft_energy')
dataset.rename_configuration_field('DFT_force', 'dft_force')
dataset.rename_configuration_field('DFT_virial', 'dft_virial')
```

3.3.5 Filtering based on XC-functional

In the Si GAP dataset, some of the data was computed using a PBE functional, and some was computed using a PW91 functional. This information is stored in the `xc_functional` field of the `Configuration.info` array.

```
set(dataset.get_configuration_field('xc_functional'))
# output: {'PBE', 'PW91'}
```

A user may want to only work with subsets of the data that were computed with the exact same DFT settings. To facilitate this, we break the original Dataset into three separate datasets using the *filter()* function (see *Filtering a Dataset* for more details).

```
no_xc_data = dataset.filter(
    'configurations',
```

(continues on next page)

(continued from previous page)

```
    lambda c: c.info.get('xc_functional', None) is None
)

pbe_data = dataset.filter(
    'configurations',
    lambda c: c.info.get('xc_functional', None) == 'PBE'
)

pw91_data = dataset.filter(
    'configurations',
    lambda c: c.info.get('xc_functional', None) == 'PW91'

)
```

CHAPTER

FOUR

USAGE

- have all of the special dictionaries described somewhere.

4.1 Property map

Note that the keys should be one of the following:

1. The name of an OpenKIM Property Definition from the list of approved OpenKIM Property Definitions
2. The name of a locally-defined property (see *Custom properties*)
3. The string `'default'`. Note that `'default'` means that an existing property will be used with support for basic fields like 'energy', 'forces', and 'stress'.

'field' is used to specify the key for extracting the property from `Configuration.info()` or `Configuartion.arrays()`. *`parse_data()`* will extract the fields provided in property_map from the configuration, and store them as Property objects in the dataset.data list. If a custom property is used, the `custom_definitions` dictionary of a Dataset must be updated to either point to the local EDN file or a Python dictionary representation of the contents of the EDN file. See *Custom properties* for more details

4.2 Applying Configuration labels

- Point to Si_PRX_GAP for an example of how to add configuration labels and build configuration sets.

4.3 Custom properties

- Point to the QM9 for an example of how to use custom properties
- Update the link in basic_example to piont to this section

4.4 Reading/writing Datasets with Markdown

4.5 Visualization

4.6 Filtering a Dataset

4.7 Checking for subsets

4.8 Data transformation

4.9 Train/test splits

4.10 Loading data

- XYZ, CFG, JSON, HDF5

4.11 Building configuration sets

CHAPTER

FIVE

CLASSES

5.1 Core classes

A *Configuration* stores all of the information about an atomic structure. The *Configuration* class inherits from the ase.Atoms class, and populates some required fields in its Atoms.info dictionary.

The most common use-case for building *Configuration* objects is to use the *load()* method of a colabfit.tools.BaseConverter instance, which will call *Configuration.from_ase()* on an existing ase.Atoms object.

```
from colabfit.tools.converters import EXYZConverter

converter = EXYZConverter()

configurations = converter.load(...)
```

5.1.1 Configuration

class colabfit.tools.configuration.**Configuration**(*labels=None*, *constraints=None*, **args*, ***kwargs*)

A Configuration is an extension of an ase.Atoms object that is guaranteed to have the following fields in its info dictionary:

- ATOMS_ID_FIELD
- ATOMS_NAME_FIELD
- ATOMS_LABELS_FIELD
- ATOMS_CONSTRAINTS_FIELD

Constructs a Configuration. Calls ase.Atoms.__init__(), then populates the additional required fields.

__hash__()

Generates a hash for self by hashing its length, constraints, positions, (atomic) numbers, simulation cell, and periodic boundary conditions

__init__(*labels=None*, *constraints=None*, **args*, ***kwargs*)

Constructs a Configuration. Calls ase.Atoms.__init__(), then populates the additional required fields.

classmethod from_ase(*atoms*)

Generates a *Configuration* from an ase.Atoms object.

A *Property* is usually extracted from a *Configuration* object using the *parse_data()* of a *Dataset()* object.

5.1.2 Property

class colabfit.tools.property.**Property**(*name*, *configurations*, *property_map*, *settings=None*, *edn=None*, *instance_id=1*, *convert_units=False*)

A Property is used to store the results of some kind of calculation or experiment, and should be mapped to an OpenKIM Property Definition. Best practice is for the Property to also point to one or more PropertySettings objects that fully define the conditions under which the Property was obtained.

edn

A dictionary defining an OpenKIM Property Instance in EDN format. For more details, see the OpenKIM Property Framework documentation.

Type dict

property_map

key = a string that can be used as a key like *self.edn[key]* value = A sub-dictionary with the following keys:

- **field:** A field name used to access `Configuration.info` or `Configuration.arrays`
- **units:** A string matching one of the units names in ase.units. These units will be used to convert the given units to eV, Angstrom, a.m.u., Kelvin, … For compound units (e.g. "eV/Ang"), the string will be split on '*' and '/'. The raw data will be multiplied by the first unit and anything preceded by a '*'. It will be divided by anything preceded by a '/'.

Type dict

configurations

A list of *`Configuration`* objects.

Type list

settings

A *`PropertySettings`* object defining the conditions under which the propoerty was obtained. This is allowed to be None, but it is highly recommended that it be provided.

Type *PropertySettings*

Parameters

- **name** (*str*) – Short OpenKIM Property Definition name
- **configurations** (*list*) – A list of ColabFit Configuration object
- **property_map** (*dict*) – A property map as described in the Property attributes section.
- **settings** (PropertySettings) – A *colabfit.property.PropertySettings* objects specifying how to compute the property.
- **edn** (*dict*) – A dictionary defining an OpenKIM Property Instance in EDN format.
- **instance_id** (*int*) – A positive non-zero integer
- **convert_units** (*bool*) – If True, converts units to those expected by ColabFit. Default is False

__delitem__(*k*)

Delete self[key].

__eq__(*other*)

Returns False if any of the following conditions are true:

- Properties point to settings with different calculation methods

- Properties point to different configurations
- OpenKIM EDN fields differ in any way

__getitem__(*k*)
Overloaded dict.__getitem__() for getting the values of self.edn

__hash__()
Hashes the Property by hashing its linked PropertySettings, Configurations, and EDN.

__init__(*name*, *configurations*, *property_map*, *settings=None*, *edn=None*, *instance_id=1*, *convert_units=False*)

Parameters

- **name** (*str*) – Short OpenKIM Property Definition name
- **configurations** (*list*) – A list of ColabFit Configuration object
- **property_map** (*dict*) – A property map as described in the Property attributes section.
- **settings** (PropertySettings) – A *colabfit.property.PropertySettings* objects specifying how to compute the property.
- **edn** (*dict*) – A dictionary defining an OpenKIM Property Instance in EDN format.
- **instance_id** (*int*) – A positive non-zero integer
- **convert_units** (*bool*) – If True, converts units to those expected by ColabFit. Default is False

__repr__()
Return repr(self).

__setitem__(*k*, *v*)
Overloaded dict.__setitem__() for setting the values of self.edn

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

convert_units()
For each key in self.property_map, convert self.edn[key] from its original units to the expected ColabFit-compliant units.

classmethod from_definition(*name*, *definition*, *conf*, *property_map*, *settings=None*, *instance_id=1*, *convert_units=False*)
Custom properties shouldn't have to satisfy the OpenKIM requirements

get_data(*k*)

Returns self[k]['source-value'] if k is a valid key, else np.nan.

Return type data (np.array or np.nan)

keys()
Overloaded dictionary function for getting the keys of self.edn

It is best practice to attach a *PropertySettings* object to a *Property* instance in order to better document the conditions under which the property was computed. This would often include information such as the DFT software package, a description of the calculation, and an example file for running the calculation.

5.1.3 PropertySettings

class colabfit.tools.property_settings.**PropertySettings**(*method=''*, *description=''*, *files=None*, *labels=None*)

This class is used to store information useful for reproducing a Property.

method
A short string describing the method used for computing the properties (e.g., 'VASP', 'QuantumEspresso', 'experiment', ...)

Type str

description
A human-readable description of the settings.

Type str

files
A list of strings, where each entry is the path to a file that may be useful for computing one or more of the properties.

Type list

labels
A list of strings; generated by parsing *files*, *description*, and *method*.

Type list

__eq__(*other*)
Equality check just compares the calculation method

__hash__()
Only hashes `self.method` for now

__init__(*method=''*, *description=''*, *files=None*, *labels=None*)

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

5.1.4 ConfigurationSet

class colabfit.tools.configuration_sets.**ConfigurationSet**(*configurations*, *description*)

configurations
A list of ase.Atoms objects

Type list

description
Human-readable metadata describing the configuration set.

Type str

labels
A list of strings; generated by making a set from the list of all labels on the configurations. Used to improve queries.

Type list

labels_counts
A list of integers of how many times each label appears in the configurations. Matches order of *labels*.

Type list

elements
A list of strings of element names present in the collection

Type list

elements_ratios
A list of floats; the total concentration of each element, given as a fraction of the total number of atoms in the collection

Type list

chemical_systems
A list of strings of chemical systems present in the collection

Type list

n_sites
The total number of atoms in the collection

Type int

__init__(*configurations*, *description*)

__repr__()
Return repr(self).

__str__()
Return str(self).

__weakref__
list of weak references to the object (if defined)

5.1.5 Dataset

class colabfit.tools.dataset.**Dataset**(*name=''*, *authors=None*, *links=None*, *description=''*, *configurations=None*, *data=None*, *property_map=None*, *configuration_label_regexes=None*, *configuration_set_regexes=None*, *property_settings_regexes=None*)

__hash__ = None

__init__(*name=''*, *authors=None*, *links=None*, *description=''*, *configurations=None*, *data=None*, *property_map=None*, *configuration_label_regexes=None*, *configuration_set_regexes=None*, *property_settings_regexes=None*)

add_configurations(*configurations*)

aggregate_metadata(*verbose=False*)

apply_transformation(*field_name*, *tform*)

Parameters

- **field_name** (*str*) – The property field name to applyl the transformation to
- **tform** (*callable*) – A BaseTransform object or a lambda function. If a lambda function is supplied, it must accept a 2-tuple as input, where the first value will be the property field data, and the second value will be the list of configurations linked to the property.

attach_configuration_labels(*fxn*, *labels*, *regex*)

Parameters

- **fxn** (*callable*) – A function that is called for every item in *self.configurations*, returning True if the PSO should be applied to the data entry.
- **labels** (*str or list*) – The labels to be applied
- **regex** (*str*) – The string that will be used for regex matching to identify the updated configurations in the future. *regex* will be appended to the name of each matching configuration

attach_dataset(*dataset*, *supersede_existing=False*)

Parameters

- **dataset** (Dataset) – The new dataset to be added
- **supersede_existing** (*bool*) – If True, any new data that is being added will be used to overwrite existing duplicate data when calling clean() or merge(). This is important for preserving Property metadata. Default is False.

attach_property_settings(*fxn*, *pso*, *regex*)

Parameters

- **fxn** (*callable*) – A function that is called for every item in *self.data*, returning True if the PSO should be applied to the data entry.
- **pso** (PropertySettings) – A property settings object
- **regex** (*str*) – The string that will be used for regex matching to identify the updated data entries in the future. *regex* will be appended to the name of each matching configuration

check_if_is_parent_dataset()

clean(*verbose=False*)
Uses hashing to compare the configurations of all properties and check for duplicate configurations.

clear_config_labels()

property configuration_label_regexes

property configuration_set_regexes

convert_units()
Converts the dataset units to the provided type (e.g., 'OpenKIM'

dataset_from_config_sets(*cs_ids*, *exclude=False*, *verbose=False*)
Returns a new dataset that only contains the specified configuration sets.

Parameters

- **cs_ids** (*int or list*) – The index of the configuration set(s) to use for building the new dataset. If *self* is a parent dataset, then *cs_ids* shoud either be a 2-tuple or a list of 2-tuples (i, j), where the configuration sets will be indexed as *dataset.data[i].configuration_sets[j]*.
- **exclude** (*bool*) – If False, builds a new dataset using all of the configuration sets _except_ those specified by *cs_ids*. Default is False.
- **verbose** (*bool*) – If True, prints progress. Default is False

Returns The new dataset. If *self* is a parent dataset, then the new dataset will also be a parent dataset.

Return type ds (*Dataset*)

define_configuration_set(*fxn*, *desc*, *regex*)

Parameters

- **fxn** (*callable*) – A function that is called for every item in *self.configurations*, returning True if the PSO should be applied to the data entry.
- **desc** (*str*) – The description of the new configuration set.
- **regex** (*str*) – The string that will be used for regex matching to identify the updated configurations in the future. *regex* will be appended to the name of each matching configuration

delete_config_label_regex(*regex*)

filter(*filter_type*, *filter_fxn*, *copy=False*, *verbose=False*)

A helper function for filtering on a Dataset. A filter is specified by providing a *filter_type* and a *filter_fxn*. In the case of a parent dataset, the filter is applied to each of the children individually.

Examples:

```
# Filter based on configuration name
regex = re.compile('example_name.*')

filtered_dataset = dataset.filter(
    'configurations',
    lambda c: regex.search(c.info[ATOMS_NAME_FIELD])
)

# Filter based on maximum force component
import numpy as np

filtered_dataset = dataset.filter(
    'data',
    lambda p: np.max(np.abs(p.edn['unrelaxed-potential-forces']['source-value
↪'])) < 1.0
)
```

Parameters

- **filter_type** (*str*) – One of 'configurations' or 'data'.

  If *filter_type* == *'configurations'*: Filters on configurations, and returns a dataset with only the configurations and their linked properties.

  If *filter_type* == *'data'*: Filters on properties, and returns a dataset with only the properties and their linked configurations.

- **filter_fxn** (*callable*) – A callable function to use as *filter(filter_fxn)*.
- **copy** (*bool*) – If True, deep copies all dataset attributes before returning filtered results. Default is False.

Returns A Dataset object constructed by applying the specified filter, extracting any objects linked to the filtered object, then copying over *property_map*, *configuration_label_regexes*, *configuration_set_regexes*, and *property_settings_regexes*.

Return type dataset (*Dataset*)

flatten()

Pseudocode:

- convert everything to the same units
- merge authors/links
- **warn if overlap (maybe this should be done in attach()?)**
  - tell me which dataset has an overlap with which other (disjoint)
  - optionally merge subset datasets
- check if conflicting CO labels, CS regexes, or PS regexes

classmethod from_markdown(*html_file_path*, *convert_units=False*, *verbose=False*)
Loads a Dataset from a markdown file.

get_available_configuration_fields()

get_configuration_field(*field*)

get_data(*property_field*, *cs_ids=None*, *exclude=False*, *concatenate=False*, *ravel=False*)
Returns a list of properties obtained by looping over *self.data* and extracting the desired field if it exists on the property.

Note that if the field does not exist on the property, that property will be skipped. This means that if there are multiple properties linked to a single configuration, *len(get_data(...))* will not be the same as *len(self.configurations)*

Parameters

- **property_field** (*str*) – The string key used to extract the property values from the Property objects
- **cs_ids** (*int or list*) – The integers specifying the configuration sets to obtain the data from. Default is None, which returns data from all configuration sets.

  If *self* is a base dataset, then *cs_ids* should be a list of integers used for indexing *self.configuration_sets*.

  If *self* is a parent dataset, then *cs_ids* should be a list of 2-tuples *(i, j)* where a configuration set will be indexed using *self.data[i].configuration_sets[j]*.
- **exclude** (*bool*) – Only to be used when *cs_ids* is not None. If *exclude==True*, then data is only returned for the configuration sets that are _not_ in *cs_ids*.
- **concatenate** (*bool*) – If True, calls np.concatenate() on the list before returning. Default is False.
- **ravel** (*bool*) – If True, calls np.concatenate() on the list before returning. Default is False.

Returns a list of Numpy arrays that were constructed by calling [np.atleast_1d(d[property_field]['source-value']) for d in self.data]

get_statistics(*property_field*)
Builds an list by extracting the values of *property_field* for each entry in the dataset, wrapping them in a numpy array, and concatenating them all together. Then returns statistics on the resultant array.

Returns

results (dict)::

..code-block:: {'average': np.average(data), 'std': np.std(data), 'min': np.min(data), 'max': np.max(data), 'average_abs': np.average(np.abs(data))}

isdisjoint(*other*, *configurations_only=False*)

issubset(*other*, *configurations_only=False*)

issuperset(*other*, *configurations_only=False*)

merge(*other*, *clean=False*)
Merges the new and current Datasets. Note that the current data supersedes any incoming data. This means that if incoming data points to an existing configuration, the incoming data is not added to the Dataset because it is assumed that the existing data pointing to the same configuration is the more important version.

The following additional changes are made to the incoming data:

- Configurations are renamed to prepend the name of their datasets
- All regexes are renamed as f"^{other.name}_.*{regex}"

Parameters

- **other** (Dataset) – The new dataset to be added to the existing one.
- **clean** (*bool*) – If True, checks for duplicates after merging. Default is False.

parse_data(*convert_units=False*, *verbose=False*)
Re-constructs *self.data* by building a list of Property objects using *self.property_map* and *self.configurations*. Modifies *self.data* in place. If *convert_units==True*, then the units in *self.property_map* are also updated.

plot_histograms(*fields=None*, *xscale='linear'*, *yscale='linear'*)
Generates histograms of the given fields.

print_configuration_sets()

property property_settings_regexes

refresh_config_labels(*verbose=False*)
Re-applies labels to the *ase.Atoms.info[ATOMS_LABELS_FIELD]* list. Note that this overwrites any existing labels on the configurations.

refresh_config_sets(*verbose=False*)
Re-constructs the configuration sets.

refresh_property_map(*verbose=False*)

refresh_property_settings(*verbose=False*)
Refresh property pointers to PSOs by matching on their linked co names

rename_configuration_field(*old_name*, *new_name*)
Renames fields in Configuration.info and Configuration.arrays.

Parameters

- **old_name** (*str*) – the original name of the field
- **new_name** (*str*) – the new name of the field

Returns None. Modifies configuration fields in-place.

rename_property(*old_name*, *new_name*)
Renames old_name field to new_name in each Property

resync(*verbose=False*)

summary()

to_markdown(*base_folder*, *html_file_name*, *data_file_name*, *data_format*, *name_field='_name'*)
Saves a Dataset and writes a properly formatted markdown file. In the case of a Dataset that has child Dataset objects, each child Dataset is written to a separate sub-folder.

Parameters

- **base_folder** (*str*) – Top-level folder in which to save the markdown and data files
- **html_file_name** (*str*) – Name of file to save markdown to
- **data_file_name** (*str*) – Name of file to save configuration and properties to
- **data_format** (*str*) – Format to use for data file. Default is 'xyz'
- **name_field** (*str*) – The name of the field that should be used to generate configuration names

train_test_split(*train_frac*)

colabfit.tools.dataset.**load_data**(*file_path*, *file_format*, *name_field*, *elements*, *default_name=''*, *labels_field=None*, *reader=None*, *glob_string=None*, *verbose=False*, ***kwargs*)
Loads configurations as a list of ase.Atoms objects.

Parameters

- **file_path** (*str*) – Path to the file or folder containing the data
- **file_format** (*str*) – A string for specifying the type of Converter to use when loading the configurations. Allowed values are 'xyz', 'extxyz', 'cfg', or 'folder'.
- **name_field** (*str*) – Key name to use to access *ase.Atoms.info[<name_field>]* to obtain the name of a configuration one the atoms have been loaded from the data file. Note that if *file_format == 'folder'*, *name_field* will be set to 'name'.
- **elements** (*list*) – A list of strings of element types
- **default_name** (*list*) – Default name to be used if *name_field==None*.
- **labels_field** (*str*) – Key name to use to access *ase.Atoms.info[<labels_field>]* to obtain the labels that should be applied to the configuration. This field should contain a comma-separated list of strings
- **reader** (*callable*) – An optional function for loading configurations from a file. Only used for *file_format == 'folder'*
- **glob_string** (*str*) – A string to use with *Path(file_path).rglob(glob_string)* to generate a list of files to be passed to *self.reader*. Only used for *file_format == 'folder'*.
- **verbose** (*bool*) – If True, prints progress bar.

All other keyword arguments will be passed with *converter.load(..., **kwargs)*

5.2 Additional tools

5.2.1 Converter

class colabfit.tools.converters.**BaseConverter**

A Converter is used to load a list of `ase.Atoms` objects and convert them to *`Configuration`* objects.

load(*file_path*, *name_field*, *elements*, *default_name=''*, *labels_field=None*, *verbose=False*, ***kwargs*)

Loads a list of `Configuration` objects.

Parameters

- **file_path** (*str*) – The path to the data files.
- **name_field** (*str*) – The key for accessing the `info` dictionary of a Configuration object to return the name of the Configuration.
- **elements** (*list*) – A list of strings of element names. Order matters or file types where a mapping from atom number to element type isn't provided (e.g., CFG files).
- **default_name** (*str*) – The name to attach to the Configuration object if `name_field` does not exist on `Configuration.info`. Default is an empty string.
- **labels_field** (*str*) – The key for accessing the `info` dictionary of a Configuration object that returns a set of string labels.
- **verbose** (*bool*) – If True, prints the loading progress. Default is False.

class colabfit.tools.converters.**CFGConverter**

A Converter for the CFG files used by the Moment Tensor Potential software

class colabfit.tools.converters.**EXYZConverter**

A Converter for Extended XYZ files

class colabfit.tools.converters.**FolderConverter**(*reader*)

This converter serves as a generic template from loading configurations from collections of files. It is useful for loading from storage formats like JSON, HDF5, or nested folders of output files from DFT codes.

Parameters **reader** (*callable*) – A function that takes in a file path and returns an *ase.Atoms* object with the relevant data in *atoms.info* and *atoms.arrays*.

__init__(*reader*)

Parameters **reader** (*callable*) – A function that takes in a file path and returns an *ase.Atoms* object with the relevant data in *atoms.info* and *atoms.arrays*.

_load(*file_path*, *name_field*, *elements*, *default_name*, *labels_field*, *verbose*, *glob_string*, ***kwargs*)

Arguments are the same as for other converters, but with the following changes:

file_path (str): The path to the parent directory containing the data files.

glob_string (str): A string to use with *Path(file_path).rglob(glob_string)* to generate a list of files to be passed to *self.reader*

All additional kwargs will be passed to the reader function as `self.reader(..., **kwargs)`

5.2.2 Transform

class colabfit.tools.transformations.**BaseTransform**(*tform*)
A Transform is used for processing raw data before loading it into a Dataset. For example for things like subtracting off a reference energy or extracting the 6-component version of the cauchy stress from a 3x3 matrix.

__init__(*tform*)

class colabfit.tools.transformations.**SubtractDivide**(*sub*, *div*)
Adds a scalar to the data, then divides by a scalar

__init__(*sub*, *div*)

class colabfit.tools.transformations.**PerAtomEnergies**
Divides the energy by the number of atoms

__init__()

class colabfit.tools.transformations.**ReshapeForces**
Reshapes forces into an (N, 3) matrix

__init__()

class colabfit.tools.transformations.**Sequential**(**args*)

__init__(**args*)

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

Symbols

A

B

C

D

E

F

G

K

L

M

N

P

R

S

T