featurebox

Release 0.0.993

boliqq07

May 06, 2023

CONTENTS:

CHAPTER

ONE

INTRODUCTION

Featurebox contains some tools (**Generation** and **Selection**) for material features. **Generation** is used for feature generation in batch model. **Selection** is used for feature selection.

And one **Extractor** in command line mode is add to obtain some special properties in batch model. The special properties need certain subsequent computational processing or third-party software participation.

In total, `Batching` is the central idea of this module. All works are for convenient data manipulation.

1.1 Generation tools

Name	Application
featurebox.featurizers.atom.mapper	`atom` Getting each element data of compound.
featurebox.featurizers.envir	`bond` Getting local environment data of compound.
featurebox.featurizers.state	`state` Getting holistic compound data.
`featurebox.featurizers.bond.expander`	Tools to transforming pure bond data.
featurebox.featurizers.batch_feature	A built-in goofy tool for generating features.
`featurebox.data.namesplit.NameSplit`	Dividing compound names to elemental proportion table.
featurebox.data.mp_access.MpAccess	Getting data from pymatgen conveniently.

All the **Generation** tools with `convert` method for single case. and `fit_transform` methods for case list.

Guide: *Data Type for Generation*

1.2 Binding selection tools

Name	Application
featurebox.selection.backforward.BackForward	Backforward selection
featurebox.selection.corr.Corr	Correlation selection.
featurebox.selection.exhaustion.Exhaustion	Exhaustion selection.
featurebox.selection.ga.GA	Genetic algorithm selection.

All the **Selection** tools are `sklearn-type`, with `fit`, `fit_transform` methods .etc.

Note: Where the binding means treat the binding features as one feature. And the binding features are selected or deleted synchronously.

Guide: *Binding Selection*

1.3 Property batching extractor

Name	Application
featurebox.cli.vasp_bader	Bader Charge
featurebox.cli.vasp_cohp	COHP
featurebox.cli.vasp_dbc	band center
featurebox.cli.vasp_dos	DOS for plot
featurebox.cli	**More** ...

All the **Extractor** tools with `convert` method for single case. and `fit_transform` methods for case list.

Guide: *Command Mode for Extractor*

Note: The properties batching extractor are suggested to use `Command line mode` . But interactive model is still available for more customized operation.

Note: The **Graph neural network** have been removed to `pyg_extension` package, which employ **envir**, **bond** and **atom** .etc to build input data.

CHAPTER

TWO

INSTALL

2.1 Method 1

Install with pip

```
pip install featurebox
```

Note: If VC++ needed for windows, Please download the dependent packages from Python Extension Packages and install offline. Such as `Spglib`. And try again, or reference to Method 2.

2.2 Method 2

Requirements Packages:

Dependence	Name	Version
necessary	sympy	>=1.6
necessary	deap	>=1.3.1
necessary	scikit-learn	>=0.22.1
necessary	torch	>=1.5.0
necessary	ase	
necessary	pymatgen	
recommend	scikit-image	
recommend	minepy	
recommend	torch_geometric	

Install by step:

1. sympy

```
pip install sympy>=1.6
```

Reference: https://www.sympy.org/en/index.html

2. deap

```
pip install deap
```

Reference: https://github.com/DEAP/deap

3. pymatgen

```
conda install --channel conda-forge pymatgen
```

Reference: https://github.com/materialsproject/pymatgen

Note: If `Spblib` needed, which need C++ to compiled, please download Python Extension Packages and pip install locally.

Such as

```
pip install /your/local/path/spglib-1.16.1-cp38-cp38-win_amd64.whl
```

4. scikit-learn

```
conda install sklearn
```

Reference: https://github.com/materialsproject/pymatgen

5. mgetool:

```
pip install mgetool
```

Reference: https://github.com/Mgedata/mgetool

6. featurebox:

```
pip install featurebox
```

7. ase:

```
pip install ase
```

Reference: https://wiki.fysik.dtu.dk/ase/ , not necessary, just for network.

CHAPTER

THREE

GUIDE

3.1 Sample Data and Background

3.1.1 Sample data

Download data from the following link: Structure List .

Usage:

```
>>> import pandas as pd
>>> from pymatgen.core import Structure
>>> structures = structure_list = pd.read_pickle("sample_data.pkl_pd")
>>> structure = structurei = structure_list[0]
```

3.1.2 Background

The `Structure` from `pymatgen` is one class to represent the crystal structure data, which contain all message of atoms and their sites. More details link: pymatgen Structure .

From this type data, we could extract atom/element names and atom/element numbers message by following code.

Such as for single case (built in `convert` function):

```
>>> structure_1 = structure_list[0]
>>> name_1 = [{str(i.symbol): 1} for i in structure_1.species]
>>> number_1 = [i.specie.Z for i in structure_1]
```

Such as for batch data (built in `transform` function):

```
>>> name_list = [[{str(i.symbol): 1} for i in si.species] for si in structure_list]
>>> number_list = [[i.specie.Z for i in si] for si in structure_list]
```

In this packages, we accept data with type like `name_list` , `number_list` as input data for `transform` . Meanwhile, The above code are built in package, thus we could accept `structure_list` as input data directly.

Note: In addition, the `ase.Atoms` could convert by Adaptor `AseAtomsAdaptor` from pymatgen or `featurebox.utils.general.AAA` . Of course, The data `name` data , `number` data could build by yourself from you code.

3.2 Data Type for Generation

Before reading this part, make sure the you have already known `Structure` from *Sample Data and Background*.

3.2.1 Definition

We divided the features into the following categories:

1. **atom/element feature** *

   The properties of atoms/elements themselves.

2. **bond features**

   The properties of inter-atomic bonds.

3. **state (overall compound) features** *

   The overall properties of the compound, include the properties that embody the overall crystal structure.

4. **crystal structure features (graph features)**

   The total of atom/element feature, bond features, state (overall compound) features.

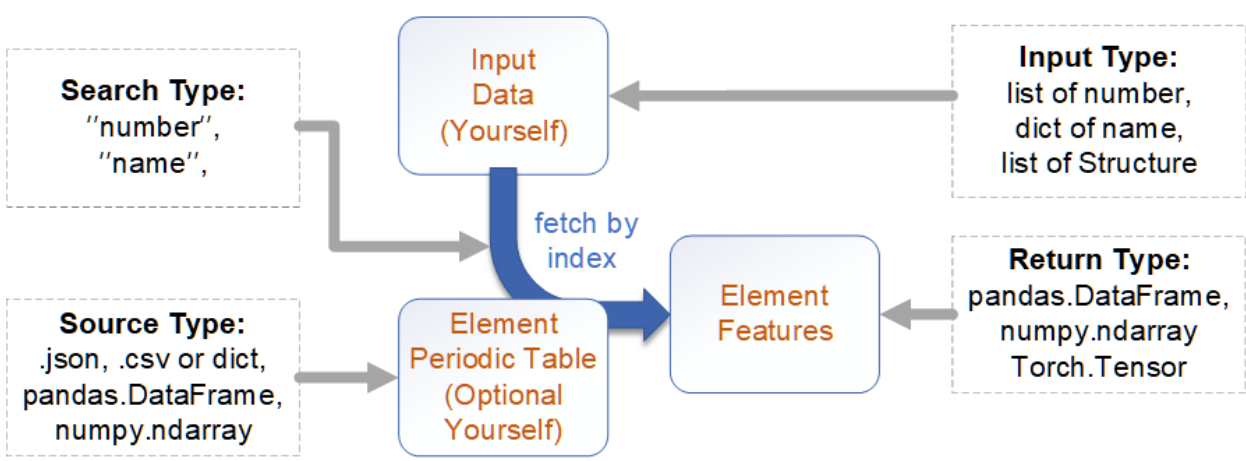
Note: Different from 1,3, The 2,4 is cannot be used directly for `sklearn`, but suit for `torch` .

3.2.2 Access

All the **Generation** tools with `convert` method for single case, and `fit_transform` methods for case list.

1. Atom/Element Features

Example Graph 1:



Example Graph 2:

Atom/Element features can be obtained by fetching periodic table data, using you input data. There are two data should offer. (at least one).

- **Your input data. The type could be atom number (`list`) or element name (`dict`) or pymatgen `Structure`.**
  (we have built-in conversion functions of `Structure`, to directly get all the atomic information in compound).
- **The element periodic table data (optional).**
  We have built-in some element periodic table ("ele_table.csv", "ie.json", "oe.csv"), To customize your element periodic table. you can offer (`.json` , `.csv`) file or any python data (`dict` , `pandas.DataFrame` , `numpy.ndarray`) in code.
  1. (`.json` , `dict`) by `AtomJsonMap` ,
  2. (`.csv`, `pandas.DataFrame` , `numpy.ndarray`) by `AtomTableMap` .
  3. And one specialized `AtomPymatgenPropMap` for fetch data from "pymatgen.core.periodic_table.json".

Example:

Input atom list

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap
>>> tmps = AtomTableMap(search_tp="number")
>>> single_sample = [1,1,1,76,76]
>>> multi_sample = [[1,1,1,76,76],[3,3,4,4]]
>>> a = tmps.convert(single_sample)
>>> b = tmps.transform(multi_sample)
```

Input element dict

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="name",return_type="np")
>>> single_sample = [{"H": 2}, {"Po": 1}]
>>> single_sample2 = {"H": 2, "Po": 1}
>>> multi_sample = [[{"H": 2}, {"Po": 1}],  [{"He": 3}, {"P": 4}]] # or
>>> multi_sample2 = [{"H": 2, "Po": 1},  {"He": 3, "P": 4}]
>>> a = tmps.convert(single_sample)
>>> a = tmps.convert(single_sample2)
>>> b = tmps.transform(multi_sample)
>>> b = tmps.transform(multi_sample2)
```

Input structure type

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="name",return_type="np")
>>> a = tmps.convert(structurei)
>>> b = tmps.transform(structure_list)
```

More:

Json Data

2. Bond Features

1. For bond features, use the structure data to extract information. The common structure data include the `Structure` of `Pymatgen`, the `Atoms` of `ase`, etc. The `Structure` and `Atoms` could mutual transform by `pymatgen.io.ase.AseAtomsAdaptor`.

3. State (overall compound) Features

There are two method to get state (overall compound) features.

- **1. Information extraction from structure data.**

Example:

```
>>> from pymatgen.core.structure import Structure
>>> structurei = Structure.from_file(r"your_path/featurebox/data/temp_test_structure/W2C.
↪cif")
```

```
>>> from featurebox.featurizers.state.state_mapper import StructurePymatgenPropMap
>>> tmps = StructurePymatgenPropMap(prop_name = ["density", "volume", "ntypesp"])
>>> a = tmps.convert(structurei)
>>> b = tmps.transform([structurei]*10)
```

where the prop_name is the name of properties of in `pymatgen` , the name of properties is not apply for all compounds, and the data could not a single number.:

```
prop_name = ["atomic_radius","atomic_mass","number","max_oxidation_state","min_oxidation_
↪state",
"row","group","atomic_radius_calculated","mendeleev_no","critical_temperature","density_
↪of_solid",
"average_ionic_radius","average_cationic_radius","average_anionic_radius",]
```

- **2. Combination or mathematical processing of atomic features according to composition ratio.**

This is one key method to get state features!!! We can get the results directly or in two step as needed.

- Get State features directly.

```
>>> from pymatgen.core.structure import Structure
>>> from featurebox.featurizers.state.statistics import WeightedAverage
>>> structurei =Structure.from_file(r"your_path/featurebox/data/W2C.cif")
```

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap
>>> data_map = AtomTableMap(search_tp="name", n_jobs=1)
>>> wa = WeightedAverage(data_map, n_jobs=1,return_type="df")
>>> x3 = [{"H": 2, "Pd": 1},{"He":1,"Al":4}]
>>> wa.fit_transform(x3)
>>> x4 = [structurei]*5
>>> wa.fit_transform(x4)
```

More combination operation `WeightedSum` , `GeometricMean` , `HarmonicMean` , `WeightedVariance` and so on can be found in *`featurebox.featurizers.state.statistics`*. More: *Pymatgen Data*.

- Get State features by step (Just for compositions with same number of atomic types).

Get the depart element feature first.

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> from featurebox.featurizers.state.union import UnionFeature
>>> from featurebox.featurizers.state.statistics import DepartElementFeature
>>> data_map = AtomJsonMap(search_tp="name",embedding_dict="ele_megnet.json", n_jobs=1)
↪# keep this n_jobs=1 and return_type="np"
>>> wa = DepartElementFeature(data_map,n_composition=2, n_jobs=1, return_type="pd")
>>> comp = [{"H": 2, "Pd": 1},{"He":1, "Al":4}]
>>> wa.set_feature_labels(["fea_{}".format(_) for _ in range(16)]) # 16 this the feature␣
↪number of built-in "ele_megnet.json"
>>> couple_data = wa.fit_transform(comp)
```

Union the depart element feature.

```
>>> # couple_data is the pd.Dataframe table.
>>> # comp is the atomic ratio of composition.
>>> uf = UnionFeature(comp,couple_data,couple=2,stats=("mean","maximum"))
>>> state_data = uf.fit_transform()
```

Note: The `UnionFeature` also could be used for your own table data!

Addition:

There one state features transformer to get Polynomial extension for table.

```
>>> import numpy as np
>>> from featurebox.featurizers.state.union import PolyFeature
>>> state_features = np.array([[0,1,2,3,4,5],[0.422068,0.360958,0.201433,-0.459164,-0.
↪064783,-0.250939]]).T
>>> state_features = pd.DataFrame(state_features,columns=["f1","f2"],index= ["x0","x1",
↪"x2","x3","x4","x5"])
>>> pf = PolyFeature(degree=[1,2])
>>> pf.fit_transform(state_features)
```

More:

Polynomial Combination, *Combination*.

3.3 Binding Selection

For material problems, some features may need to appear together to make sense. such as the hole radius and atomic radius of inner atoms.

Or we use the depart element features to model the composition properties directly. rather than combine element features, and filter. such as A-site properties and B-site properties in AB2-type compound should be selected and dropped simultaneously.

All the selection method of featurebox are with `Binding`, of course, if without setting any bindings, they can degenerate to normal algorithms.

Example:

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.svm import SVR
>>> from featurebox.selection.backforward import BackForward
>>> X,y = load_boston(return_X_y=True)
>>> svr= SVR()
>>> bf = BackForward(svr, random_state=1, multi_index=[0,8], multi_grade=2)
>>> new_x = bf.fit_transform(X,y)
>>> bf.support_
```

The `multi_index` parameter means the bonded primary feature index.

All the selection tools are `sklearn-type`, with `fit`, `fit_transform` methods .etc.

More:

featurebox.selection.backforward.BackForward

featurebox.selection.corr.Corr

featurebox.selection.exhaustion.Exhaustion

featurebox.selection.ga.GA

3.4 Command Mode for Extractor

The Extractor include:

`bandgap`, `dbc`, `bader`, `cohp`, `dos`, `general`, `diff`, `converge`

3.4.1 Using

1. Run in command line mode (suggested). All message (help) could get by `'-h'` .

Examples:

```
$ featurebox bandgap -h

$ fbx bandgap -h

$ featurebox bandgap -p /home/parent_dir

$ featurebox bandgap -f /home/parent_dir/paths.temp
```

Use `fbx -h` or `fbx {sub_cmd} -h` for more details.

2. Run in python for more customization.

```
>>> from featurebox.cli.vasp_dos import DosxyzPathOut
>>> dosxyz = DosxyzPathOut(n_jobs=4, store_single=True)
>>> result = dosxyz.transfrom(paths_list)
>>> # More part: The following is not in command model.
>>> # final treatment to extractor need message and formatting.
>>> features = dosxyz.extractor(result,atoms=[0, 1, 2, 3], ori=["p-x","d-xy"],format_
↪path=None)
```

3.4.2 Key

1. All the sub-extractor such as 'bader' are offer 2 input method in command mode.

For single case:

If one path is offered by `-p` for `Single Case` (default, and in WORKPATH), please make sure the necessary files under the path exists.

For batching:

If paths is offered by `-f` for `Batching`, please make sure the file such as 'paths.temp' exits and not empty.

Generated one file by `findpath` command in mgetool package is suggest. use `findpath` command directly now, to get all sub-folder in current path.

Or use `findpath -h` for more help.

2. Some Extractor tools are need third-party tools. please download and installed them in advance.

Property	Name
dbc	vaspkit <=1.2.1
cohp	lobster
bader	bader
bader	chgsum.pl
chg_diff	chgdiff.pl

CHAPTER

FOUR

FEATUREBOX

4.1 featurebox package

4.1.1 Subpackages

featurebox.cli package

The `cli` part include:

`bandgap`, `dbc`, `bader`, `cohp`, `dos`, `general`, `diff`, `converge`

1. Run in command line mode (suggested). All message (help) could get by `'-h'` .

Examples:

```
$ featurebox bandgap -h

$ fbx bandgap -h

$ featurebox bandgap -p /home/parent_dir

$ featurebox bandgap -f /home/parent_dir/paths.temp
```

2. Run in python.

```
>>> from featurebox.cli.vasp_dos import DosxyzPathOut
>>> dosxyz = DosxyzPathOut(n_jobs=4, store_single=True)
>>> result = dosxyz.transform(paths_list)
```

Submodules

featurebox.cli.vasp_bader module

class `featurebox.cli.vasp_bader.`**`BaderStartInter`**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: *`BaderStartZero`*

Get bader from paths and return csv file. For some system can't run BaderStartZero.

Download bader from http://theory.cm.utexas.edu/henkelman/code/bader/

Download chgsum.pl from http://theory.cm.utexas.edu/vtsttools/download.html

1. Copy follow code to form one 'badertoACF.sh' file, and 'sh badertoACF.sh':

Notes:

```
#!/bin/bash

old_path=$(cd "$(dirname "$0")"; pwd)

for i in $(cat paths.temp)

do

cd $i

echo $(cd "$(dirname "$0")"; pwd)

chgsum.pl AECCAR0 AECCAR2

bader CHGCAR -ref CHGCAR_sum

cd $old_path

done
```

2. tar -zcvf data.tar.gz ACF.dat POTCAR POSCAR.
3. Move to other system and run 'tar -zxvf data.tar.gz'.
4. Run with this class (-j 1).

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_bader.**BaderStartSingleResult**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: *BaderStartZero*

Get bader from paths and return csv file. Avoid Double Calculation. Just reproduce the 'results_all' from a 'result_single' files. keeping the 'result_single.csv' files exists.

read(*path*, ***kwargs*)

Run linux cmd and return result, make sure the bader is installed.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_bader.**BaderStartZero**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: _BasePathOut

Get bader from paths and return csv file.

1.Download bader from http://theory.cm.utexas.edu/henkelman/code/bader/

2. Download chgsum.pl from http://theory.cm.utexas.edu/vtsttools/download.html

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

static extract(*data: DataFrame*, *atoms*, *format_path: Optional[Callable] = None*)

The last process! Extract the message in data, and formed it to be one table or ML or plot.

Parameters

- **data** (*pd.DateFrame*) – to transform data.
- **format_path** (*Callable*) – function to deal with each path, for better shown.

Returns
res_data – extracted and formed data.

Return type
pd.DateFrame

static read(*path*, *store=False*, *store_name='bader_single.csv'*)

Run linux cmd and return result, make sure the bader is installed.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.cli.vasp_bgp module

class featurebox.cli.vasp_bgp.**BandGapPy**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: _BasePathOut

Get band gap from vasprun.xml by pymatgen.

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_bgp.**BandGapStartInter**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: *BandGapStartZero*

For some system can't run this BandGapStartZero. Download vaspkit from https://vaspkit.com/installation.html#download

1. Copy follow code to form one 'bg.sh' file, and 'sh bg.sh':

Notes:

```
#!/bin/bash

old_path=$(cd "$(dirname "$0")"; pwd)

for i in $(cat paths.temp)

do
```

(continues on next page)

(continued from previous page)

```
cd $i

echo $(cd "$(dirname "$0")"; pwd)

vaspkit -task 911 > BAND_GAP

cd $old_path

done
```

2. tar -zcvf data.tar.gz BAND_GAP.
3. Move to other system and run 'tar -zxvf data.tar.gz'.
4. Run with this class.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_bgp.**BandGapStartSingleResult**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: *BandGapStartZero*

Avoid Double Calculation. Just reproduce the 'results_all' from a 'result_single' files. keeping the 'result_single.csv' files exists.

read(*path*, ***kwargs*)

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_bgp.**BandGapStartZero**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: _BasePathOut

Get band gap from paths and return csv file. VASPKIT Version: 1.2.1 or below. Download vaspkit from https://vaspkit.com/installation.html#download

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

static extract(*data*, **args*, *format_path: Optional[Callable] = None*, ***kwargs*)

Extract the message in data, and formed it.

Parameters

- **data** (*pd.DateFrame*) – transformed data.
- **format_path** (*Callable*) – function to deal with each path, for better shown.

Returns

res_data – extracted and formed data.

Return type

pd.DateFrame

static read(*path*, *store=False*, *store_name='bgp_kit_single.csv'*)

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.cli.vasp_chg_diff module

Due to the CHGCAR file are large, We don't use python code to get chg_diff. (CHGCAR2-CHGCAR1) Result = f2 - f1, make sure the rank of paths in two files are matching.

file1

p1

p2

...

file2

p1'

p2'

...

featurebox.cli.vasp_cohp module

class featurebox.cli.vasp_cohp.**COHPStartInter**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=True*)

Bases: *COHPStartZero*

For some system can't run this COHPStartZero. Download lobster from http://www.cohp.de/

1. Copy follow code to form one 'lob.sh' file, and 'sh lob.sh' (change the atoms couple):

Notes:

```
#!/bin/bash

old_path=$(cd "$(dirname "$0")"; pwd)

for i in $(cat paths.temp)

do

cd $i

echo $(cd "$(dirname "$0")"; pwd)

echo COHPStartEnergy -10 > lobsterin

echo COHPEndEnergy 5 >> lobsterin
```

(continues on next page)

(continued from previous page)

```
echo cohpBetween atom 45 atom 31 >> lobsterin

lobster > look

sleep 20m

cd $old_path


done
```

2. tar -zcvf data.tar.gz "ICOHPLIST.lobster", "COHPCAR.lobster".
3. Move to other system and run 'tar -zxvf data.tar.gz'.
4. Run with this class.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class `featurebox.cli.vasp_cohp.`**COHPStartSingleResult**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: *COHPStartInter*

Avoid Double Calculation. Just reproduce the 'results_all' from a 'result_single' files. keeping the 'result_single.csv' files exists.

read(*path*, ***kwargs*)

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class `featurebox.cli.vasp_cohp.`**COHPStartZero**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=True*)

Bases: `_BasePathOut`

Get d band center from paths and return csv file. Download lobster from http://www.cohp.de/

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

static read(*path*, *store=False*)

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.cli.vasp_converge module

class `featurebox.cli.vasp_converge.`**`ConvergeChecker`**(*n_jobs: int = 1, tq: bool = True, store_single=False*)

Bases: `_BasePathOut`

Get energy from paths and return csv file. VASPKIT Version: 1.2.1 or below.

`batch_after_treatment`(*paths, res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

`run`(*path: Path, files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

`featurebox.cli.vasp_converge.`**`check_convergence`**(*pt: Union[str, Path, PathLike, Path], msg=None*)

Check final energy.

Parameters

- **`pt`** – (str, path.Path, os.PathLike,pathlib.Path), path
- **`msg`** – (list of str), message.

Returns
(tuple), bool and msg list

Return type
res

featurebox.cli.vasp_dbc module

class `featurebox.cli.vasp_dbc.`**`DBCPy`**(*n_jobs: int = 1, tq: bool = True, store_single=False, method='ele'*)

Bases: `_BasePathOut`

Get d band center by pymatgen and return csv file. pymatgen>=2022.5.26

`batch_after_treatment`(*paths, res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

`run`(*path: Path, files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class `featurebox.cli.vasp_dbc.`**`DBCStartInter`**(*n_jobs: int = 1, tq: bool = True, store_single=False*)

Bases: *`DBCStartZero`*

For some system can't run this DBCStartZero.

Download vaspkit from https://vaspkit.com/installation.html#download

1. Copy follow code to form one 'dbc.sh' file, and run 'sh dbc.sh':

Notes:

```
#!/bin/bash

old_path=$(cd "$(dirname "$0")"; pwd)

for i in $(cat paths.temp)

do

cd $i

echo $(cd "$(dirname "$0")"; pwd)

vaspkit -task 503

cd $old_path

done
```

2. tar -zcvf data.tar.gz D_BAND_CENTER.
3. Move to other system and run 'tar -zxvf data.tar.gz'.
4. Run with this class.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_dbc.**DBCStartSingleResult**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: *DBCStartZero*

Avoid Double Calculation. Just reproduce the 'results_all' from a 'result_single' files. keeping the 'result_single.csv' files exists.

read(*path*, ***kwargs*)

Run linux cmd and return result, make sure the vaspkit is installed.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_dbc.**DBCStartZero**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*)

Bases: _BasePathOut

Get d band center from paths and return csv file. VASPKIT Version: 1.2.1 or below. Download vaspkit from https://vaspkit.com/installation.html#download

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

static extract(*data: DataFrame*, *atoms=None*, *ele=None*, *join_ele=None*, *format_path: Optional[Callable] = None*)

atoms start from 1. This atom number are different to the structure atom number.

```
>>> self.extract(res, ele=["O1","C1",...], join_ele=[f'{i}1' if i !='Mo' else
↪'Mo18' for i in self.doping], )
```

static read(*d*, *store=False*, *store_name='dbc_single.csv'*)

Run linux cmd and return result, make sure the vaspkit is installed.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_dbc.**DBCxyzPathOut**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*, *method='atom'*)

Bases: _BasePathOut

Get d band center by code and return csv file.

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

static extract(*data: DataFrame*, *atoms*, *orbit=None*, *format_path: Optional[Callable] = None*)

atoms start from 0.

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.cli.vasp_dbc.**get_atom_pdos_center**(*dos: Optional[CompleteDos] = None*, *mark_orbital=None*, *mark_atom_numbers=None*)

featurebox.cli.vasp_dbc.**get_ele_pdos_center**(*dos: Optional[CompleteDos] = None*, *mark_orbital=None*, *mark_element=None*)

featurebox.cli.vasp_dos module

class featurebox.cli.vasp_dos.**DosPy**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*, *method='ele'*)

Bases: _BasePathOut

Get d band center from paths and return csv file.

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

class featurebox.cli.vasp_dos.**Dosxyz**(*doscar='DOSCAR'*, *poscar='CONTCAR'*, *vasprun='vasprun.xml'*, *ispin=2*, *lmax=2*, *lorbit=11*, *spin_orbit_coupling=False*, *read_pdos=True*, *max=10*, *min=-10*)

Bases: object

Read DOSCAR and get pdos and band centor.

Create a Doscar object from a VASP DOSCAR file.

Parameters

- **doscar** (*str*) – Filename of the VASP DOSCAR file to read.
- **poscar** (*str*) – File POSCAR/CONTCAR
- **vasprun** (*str*) – File vasprun.xml
- **(optional** (*read_pdos*) – int): ISPIN flag. Set to 1 for non-spin-polarised or 2 for spin-polarised calculations. (Default = 2.)
- **(optional** – int): Maximum l angular momentum. (d=2, f=3). Default = 2.
- **(optional** – int): The VASP LORBIT flag. (Default=11).
- **(optional** – int): max value
- **(optional** – int): min value
- **(optional** – bool): Spin-orbit coupling (Default=False).
- **(optional** – bool): Set to True to read the atom-projected density of states (Default=True).

calculate(*orb='d'*, *species: Optional[List[str]] = None*, *atoms: Optional[List[int]] = None*, *emax=2*, *emin=-10*, *m=None*)

species (optional:list(str)): List of atomic species strings, e.g. ['Fe', 'Fe', 'O', 'O', 'O']. Default=None.

dbc_part_atom_num(*path=None*)

dbc_part_atom_type(*path=None*)

property number_of_channels

number_of_header_lines = 6

pdos_by_atom_num(*spin=None*)

pdos_by_atom_type(*spin=None*)

pdos_by_spdf(*atoms=None*, *spin=None*, *l=None*)

pdos_by_spdf_atom_num(*spin=None*, *l=None*)

pdos_by_spdf_atom_type(*spin=None*, *l=None*)

pdos_by_spdf_xyz_atom_num(*path=None*)

pdos_by_spdf_xyz_atom_type(*path=None*)

static pdos_column_names(*lmax*, *ispin*)

pdos_select(*atoms=None*, *spin=None*, *l=None*, *m=None*)

Returns a subset of the projected density of states array.

pdos_sum(*atoms=None*, *spin=None*, *l=None*, *m=None*)

process_header()

read_atomic_dos_as_df(*atom_number*)

read_header()

read_projected_dos()

Read the projected density of states data into

scale(*data*) → DataFrame

class featurebox.cli.vasp_dos.**DosxyzPathOut**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*, *method='ele'*)

Bases: _BasePathOut

Get dos from paths and return csv file.

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

run(*path: Path*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.cli.vasp_dos.**get_atom_pdos**(*dos: Optional[CompleteDos] = None*, *mark_orbital=None*, *mark_atom_numbers=None*, *sigma=0.1*, *path=None*)

featurebox.cli.vasp_dos.**get_ele_pdos**(*dos: Optional[CompleteDos] = None*, *mark_orbital=None*, *mark_element=None*, *sigma=0.1*, *path=None*)

featurebox.cli.vasp_general_diff module

class featurebox.cli.vasp_general_diff.**GeneralDiff**(*n_jobs: int = 1*, *tq: bool = True*, *store_single=False*, *mod='pymatgen.io.vasp'*, *cmd='Vasprun'*, *necessary_files='vasprun.xml'*, *prop='final_energy'*)

Bases: _BasePathOut2

Get data from couples of paths and return csv file.

Notes:

```
mod="pymatgen.io.vasp"         # Module to get class.
cmd="Vasprun"                  # class to get object.
necessary_files="vasprun.xml"  # class input file.
prop="final_energy"            # class.property name.
```

batch_after_treatment(*paths*, *res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

run(*paths: List[Path]*, *files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.cli.vasp_general_single module

class featurebox.cli.vasp_general_single.**General**(*n_jobs: int = 1, tq: bool = True, store_single=False, mod='pymatgen.io.vasp', cmd='Vasprun', necessary_files='vasprun.xml', prop='final_energy'*)

Bases: _BasePathOut

Get data from paths and return csv file.

Default keys Notes:

```
mod="pymatgen.io.vasp"         # Module to get class.
cmd="Vasprun"                  # class to get object.
necessary_files="vasprun.xml"  # class input file.
prop="final_energy"            # class.property name.
```

batch_after_treatment(*paths, res_code*)

4. Organize batch of data in tabular form, return one or more csv file. (force!!!).

run(*path: Path, files: Optional[List] = None*)

3.Run with software and necessary file and get data. (1) Return result in code, or (2) Both return file to each path and in code.

featurebox.data package

Data tools.

Embedded data: "ele_table.csv", "ele_megnet.json", "ie.json", "oe.csv"

Submodules

featurebox.data.check_data module

class featurebox.data.check_data.**CheckElements**(*check_method: ~typing.Union[~typing.List[str], str] = 'name', func: ~typing.Callable = <function CheckElements.<lambda>>*)

Bases: object

Check the element in available elements or not.

AVAILABLE_ELE_NUMBER:
(1~84) + (89, 90, 91, 92).

AVAILABLE_ELE_NAME:
('H'~'Bi') + ('Ac', 'Th', 'Pa', 'U').

Parameters

- **check_method** (*str*) – Check by number or name of element. Optional ("name","number")
- **func** (*callable*) – Processing for elements. Such as for element in pymatgen:

```
>>> func = lambda x: [x.Z, ]
>>> func2 = lambda x: [x.name, ]
```

Examples

```
>>> ce = CheckElements.from_list(check_method="name")
>>> ce.check(["Na","Al","Ta"])
['Na', 'Al', 'Ta']
>>> ce = CheckElements.from_list(check_method="name")
>>> ce.check([["Na","Al"],["Na","Ta"]])
[['Na', 'Al'], ['Na', 'Ta']]
>>> ce.check([["Na","Al"],["Na","Ra"],["Zn","H"]])
The 1 (st,ed,th) sample ['Na', 'Ra'] is with element out of AVAILABLE_ELE_NAME
 please to check_data.py for more information.
[['Na', 'Al'], ['Zn', 'H']]
>>> ce.passed_idx()
array([0, 2], dtype=int64)
```

Examples

```
>>> ce = CheckElements.from_pymatgen_structures()
...
```

check(*samples: List*) → List

Parameters
samples (*list*) – Names or numbers, or list of pymatgen.Structure

Returns
result – List of filtered structures.

Return type
list

classmethod from_list(*check_method='name'*, *grouped='False'*)

Get checker for list of name or number.

classmethod from_pymatgen_structures()

Get checker for list of pymatgen.Structure.

passed_idx() → ndarray

The mark for all structures, return np.ndarray index.

featurebox.data.data_sep module

class featurebox.data.data_sep.**DataSameSep**(*data: Optional[Dict] = None*, *sep='-'*, *sites_name='S'*, *dup=3*, *prefix=None*)

Bases: `object`

Settle data, dispatch data with "all" mark to each site. Make sure the values of dict are Immutable type,such as float,init. Otherwise, the stored data would change with the input data, even if later than the call of this class/function.

Examples:

```
>>> d1 = {"Ta-S1":{"bond1":3.4,"bond2":3.5},"Co-S2":{"bond1":3.2,"bond2":3.1},"Fe-
↪Sall":{"bond1":3.2,"bond2":3.1}}
>>> dss = DataSameSep(d1)
>>> dss["Ta-S1"]={"bond1":3.2,"bond2":3.5} # cover the old.
>>> dss.replace({"Ta-S1":{"bond1":3.4,"bond2":3.5},"Co-S2":{"bond1":3.2,"bond2":3.1}
↪}) # cover the old.
>>> dss.replace_entry(label="Ta",site=1,entry={"bond1":3.2,"bond2":3.5}) # cover␣
↪the old.
```

```
>>> dss.update({"Ta-S1":{"bond1":3.4,"bond2":3.5},"Co-S2":{"bond1":3.2,"bond2":3.1}}
↪) # add
>>> dss.update_entry(label="Co",site=0,entry={"bond1":3.2}) # add
>>> dss.update_entry_kv(label="Mg",site="all",key="bond1",value=3.2) # add
>>> dict_data = dss.settle()
>>> pd_data = dss.settle_to_pd(sort=True)
>>> print(pd_data)
       bond1  bond2
Co-S0    3.2    NaN
Co-S2    3.2    3.1
Fe-S0    3.2    3.1
Fe-S1    3.2    3.1
Fe-S2    3.2    3.1
Mg-S0    3.2    NaN
Mg-S1    3.2    NaN
Mg-S2    3.2    NaN
Ta-S1    3.2    3.5
```

Make sure the key of data are formatted by {label}-{Si or Sall} !!! and all values is dict type. The 'S' is the same with sites_name.

param data
first key are formated by {label}{sep}{Si or Sall}.

type data
(dict of dict)

param sep
default "-".

type sep
(str)

param sites_name
default "S".

type sites_name
(str)

param dup
default 3.

type dup
(int)

param prefix
the class prefix of one batch data.

type prefix
(str)

`replace`(*data: Dict*)

Replace dict data.

Parameters
data (*dict*) – {entry_key: entry}.

`replace_entry`(*label: str*, *site: Union[int, str]*, *entry: Dict*, *prefix=None*)

Replace entry!! This would cover the old entry.

Parameters

- **label** (*str*) – label name.
- **site** (*int*) – number small than self.dup, or "all".
- **entry** (*dict*) – entry data.
- **prefix** (*str*) – prefix name for batch of data.

`settle`(*sort=False*) → Dict

Settle data and return a formed dict.

Parameters
sort (*bool*) – sort the entry keys or not.

Returns
data_settled – new dict.

Return type
dict

`settle_to_pd`(*sort=False*) → DataFrame

Settle data and return a formed pd.Dataframe.

Parameters
sort (*bool*) – sort the entry keys or not.

Returns
data_settled – new table.

Return type
pd.Dataframe

`spilt`(*prefix_label_site=''*) → Tuple

Try to get prefix,label,site_number.

`update`(*data: Dict*)

Add dict data.

Parameters
data (*dict*) – {entry_key: entry}.

`update_entry`(*label: str*, *site: Union[int, str]*, *entry: Dict*, *prefix=None*)

Add dict data to entry.

Parameters

- **label** (*str*) – label name.
- **site** (*int*) – number small than self.dup, or "all".

- **entry** (*dict*) – entry data.
- **prefix** (*str*) – prefix name for batch of data.

update_entry_kv(*label: str*, *site: Union[int, str]*, *key: str*, *value: Any*, *prefix=None*)

Add dict data to entry.

Parameters

- **label** (*str*) – label name.
- **site** (*int*) – number small than self.dup, or "all".
- **key** (*str*) – name of property.
- **value** (*any*) – value (float, int, str)
- **prefix** (*str*) – prefix name for batch of data.

update_from_pd(*df: Union[DataFrame, str]*)

Read table and update to data. The table must be the formed by self.settle_to_pd function.

if df is str, try: df = pd.read_csv("df_name", index_col=0).T

Parameters

df (*(pd.DataFrame,str)*) –

featurebox.data.mp_access module

class featurebox.data.mp_access.**MpAccess**(*api_key: str = 'Di28ZMunseR8vr46'*)

Bases: `object`

API for pymatgen database, access pymatgen to get data.

Examples

```
>>> mpa = MpAccess("Di28ZMunseR8vr57") # change yourself key.
>>> ids = mpa.get_ids({"elements": {"$in": ["Al","O"]},'nelements': {"$lt": 2, "$gte
↪": 1}})
number 29
>>> df = mpa.data_fetcher(mp_ids=ids, mp_props=['material_id', "cif"])
Will fetch 29 inorganic compounds from Materials Project
>>> structures_list = mpa.cifs_to_structures()
...
```

Parameters

api_key (*str:*) – pymatgen key.

cifs_to_structures(*cifs: Optional[List[str]] = None*) → List[Structure]

Get structures from cifs

data_fetcher(*mp_ids: Optional[List[str]] = None*, *mp_props: Optional[List[str]] = None*, *elasticity: bool = False*) → DataFrame

Fetch file from pymatgen.

prop_name=['band_gap','density','"icsd_ids"'volume','material_id','pretty_formula','elements','"energy", 'efermi','e_above_hull','formation_energy_per_atom','final_energy_per_atom','unit_cell_formula', 'spacegroup','nelements'"nsites","final_structure","cif","piezo","diel"]

Parameters

- **mp_ids** (*list of str*) – list of MP id of pymatgen.
- **mp_props** (*list of str*) – prop_names
- **elasticity** (*bool*) – obtain elasticity or not.

Returns
properties Table.

Return type
pandas.DataFrame

get_ids(*criteria: Optional[Dict] = None*)

Search id by criteria.

support_property = ['energy', 'energy_per_atom', 'volume', 'formation_energy_per_atom', 'nsites', 'unit_cell_formula','pretty_formula', 'is_hubbard', 'elements', 'nelements', 'e_above_hull', 'hubbards', 'is_compatible', 'spacegroup', 'task_ids', 'band_gap', 'density', 'icsd_id', 'icsd_ids', 'cif', 'total_magnetization','material_id', 'oxide_type', 'tags', 'elasticity']

Examples

```
>>> from itertools import combinations
>>> name_list = ["NaCl","CaCo3"]
>>> criteria = {
... 'pretty_formula': {"$in": name_list},
... 'nelements': {"$lt": 3, "$gte": 3},
... 'spacegroup.number': {"$in": [225]},
... 'crystal_system': "cubic",
... 'nsites': {"$lt": 20},
... 'formation_energy_per_atom': {"$lt": 0},
... # "elements": {"$all": "O"},
... # "piezo":{"$ne": None}
... # "elements": {"$all": "O"},
... "elements": {"$in": list(combinations(["Al", "Co", "Cr", "Cu", "Fe", 'Ni'],␣
↪5))}}
```

where, `"$gt"` >, `"$gte"` >=, `"$lt"` <, `"$lte"` <=, `"$ne"` !=, `"$in"`, `"$nin"` (not in), `"$or"`, `"$and"`, `"$not"`, `"$nor"` , `"$all"`

get_ids_from_web_table(*path_file: Optional[str] = None*) → List[str]

This is method to read csv file download from web,the file name is '_Materials Project.csv', which contains "Materials Id" columns.

featurebox.data.namesplit module

featurebox.featurizers package

Subpackages

featurebox.featurizers.atom package

Submodules

featurebox.featurizers.atom.mapper module

Get pure atom properties.

Embedded data: "ele_table.csv", "ele_megnet.json", "ie.json", "oe.csv"

class featurebox.featurizers.atom.mapper.**AtomJsonMap**(*embedding_dict: Optional[Union[str, Dict]] = None, search_tp: str = 'auto', feature_labels=None, **kwargs*)

Bases: *BinaryMap*

Fixed Atom json map.

Examples

```
>>> tmps = AtomJsonMap(search_tp="number")
>>> s = [1,76]                    #[i.specie.Z for i in structure]
>>> a = tmps.convert(s)
>>> tmps = AtomJsonMap(search_tp="name")
>>> s = [{"H": 2, }, {"Al": 1}]  #[i.species.as_dict() for i in pymatgen_structure.
↪sites]
>>> a = tmps.convert(s)
>>>
>>> tmps = AtomJsonMap(search_tp="name")
>>> s = [[{"H": 2, }, {"Ce": 1}],[{"H": 2, }, {"Al": 1}]]
>>> a = tmps.transform(s)
```

Parameters

embedding_dict – (str,dict) Name of file or dict,element to element vector dictionary

Provides the pre-trained elemental embeddings using formation energies, which can be used to speed up the training. The embeddings are also extremely useful elemental descriptors that encode chemical similarity that may be used in other ways.

convert_dict(*atoms: List[dict]*) → ndarray

Convert atom {symbol: fraction} list to numeric features

convert_number(*atoms: List[int]*) → ndarray

convert list of number to data

class featurebox.featurizers.atom.mapper.**AtomMap**(*n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'any', **kwargs*)

Bases: *BaseFeature*

Base class for atom converter. Map the element type and weight to element data.

Parameters

- **batch_size** (*int*) – size of batch.
- **batch_calculate** (*bool*) – batch_calculate or not.
- **n_jobs** (*int*) – Parallel number.

- **on_errors** (*str*) – How to handle the exceptions in a feature calculations. Can be nan, keep, raise. When 'nan', return a column with np.nan. The length of column corresponding to the number of feature labs. The default is 'raise' which will raise up the exception.
- **return_type** (*str*) – Specific the return type. Can be any, np,``array`` and df. 'array' and 'df' force return type to np.ndarray and pd.DataFrame respectively. If 'any', without type conversion . Default is 'any'

static get_csv_embeddings(*data_name: str*) → DataFrame

get csv preprocessing

static get_json_embeddings(*file_name: str = 'ele_megnet.json'*) → Dict

get json preprocessing

class featurebox.featurizers.atom.mapper.**AtomPymatgenPropMap**(*prop_name: Union[str, List[str]], func: Optional[Callable] = None, search_tp: str = 'name', **kwargs*)

Bases: *BinaryMap*

Get pymatgen element preprocessing. prop_name = ["atomic_radius", "atomic_mass", "number", "max_oxidation_state", "min_oxidation_state", "row", "group", "atomic_radius_calculated", "mendeleev_no", "critical_temperature", "density_of_solid", "average_ionic_radius", "average_cationic_radius", "average_anionic_radius",]

Examples

```
>>> tmps = AtomPymatgenPropMap(search_tp="number",prop_name=["X"])
>>> s = [1,76]
>>> a = tmps.convert(s)
>>> tmps = AtomPymatgenPropMap(search_tp="name",prop_name=["X"])
>>> s = [{"H": 2, }, {"Po": 1}]  #[i.species.as_dict() for i in pymatgen.structure.
↪sites]
>>> a = tmps.convert(s)
>>> tmps = AtomPymatgenPropMap(search_tp="name",prop_name=["X"])
>>> s = [[{"H": 2, }, {"Po": 1}],[{"H": 2, }, {"Po": 1}]]
>>> a = tmps.transform(s)
```

Parameters

- **prop_name** – (str,list of str) prop name or list of prop name
- **func** – (callable or list of callable) please make sure the size of it is the same with prop_name.
- **search_tp** – (str) location method. "name" for dict "number" for int.

convert_dict(*atoms: List[Dict]*) → ndarray

Convert atom {symbol: fraction} list to numeric features

convert_number(*atoms: List[int]*) → ndarray

Convert int list to numeric features

property feature_labels

Generate attribute names.

Returns

([str]) attribute labels.

class featurebox.featurizers.atom.mapper.**AtomTableMap**(*tablename: Optional[Union[str, ndarray, DataFrame]] = 'oe.csv', search_tp: str = 'auto', **kwargs*)

Bases: *BinaryMap*

Fixed Atom embedding map. Default table is oe.csv. you can change the table yourself for different preprocessing. The table contains elemental features for 92 U elements at least. Please check all your data is int or float!!!

Such as:

Data	F0	F1	...
H	V	V	...
He	V	V	...
Li	V	V	...
Be	V	V	...
...	...	...	...

Examples

```
>>> tmps = AtomTableMap(search_tp="number")
>>> s = [1,76]
>>> tmps.convert(s)
array([[2.245000e+01, 0.000000e+00, 0.000000e+00, 0.000000e+00,
        0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00,
        0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00,
        0.000000e+00, 0.000000e+00, 0.000000e+00, 0.000000e+00,
        0.000000e+00, 0.000000e+00, 0.000000e+00],
       [2.383710e+03, 3.937715e+04, 3.783280e+03, 9.866700e+02,
        8.349720e+03, 6.978800e+02, 1.861780e+03, 1.549970e+03,
        9.784700e+02, 2.231900e+02, 2.633000e+02, 1.689800e+02,
        2.854000e+01, 0.000000e+00, 1.841000e+01, 0.000000e+00,
        0.000000e+00, 0.000000e+00, 0.000000e+00]])
```

```
>>> tmps = AtomTableMap(search_tp="name")
>>> s = [{"H": 2, }, {"Po": 1}]  #[i.species.as_dict() for i in pymatgen.structure.
↪sites]
>>> a = tmps.convert(s)
...
>>> tmps = AtomTableMap(search_tp="name",tablename="oe.csv")
>>> s = [[{"H": 2, }, {"Po": 1}],[{"H": 2, }, {"Po": 1}]]
>>> a = tmps.transform(s)
...
```

```
>>> tmps = AtomTableMap(tablename=None)
>>> tmps = AtomTableMap(tablename="ele_table.csv")
>>> s = [{"H": 2, }, {"Pd": 1}]
>>> b = tmps.convert(s)
...
```

Parameters

- **tablename** (*str,np.ndarray, pd.Dateframe*) – 1. Name of table in bgnet.preprocessing.resources. if tablename is None, use the embedding "ele_table.csv".
  2. np.ndarray, search_tp = "number".
  3. pd.dataframe, search_tp = "name"
- **search_tp** (*str*) – Name

convert_dict(*atoms: List[Dict]*) → ndarray

Convert atom {symbol: fraction} list to numeric features

convert_number(*atoms: List[int]*) → ndarray

Convert atom number list to numeric features

property feature_labels

Generate attribute names.

Returns
([str]) attribute labels.

static get_ele_embeddings(*name='ele_table_norm.csv'*) → DataFrame

get CSV preprocessing

class featurebox.featurizers.atom.mapper.**BinaryMap**(*search_tp: str = 'auto', weight: bool = False, **kwargs*)

Bases: *AtomMap*

Base converter with 2 different search_tp.

Parameters

- **search_tp** – (str)
- **weight** – (bool) , For true,the same key data are summed together.
- ****kwargs** –

abstract convert_dict(*d: List[Dict]*)

convert list of dict to data

abstract convert_number(*d: List[int]*)

convert list of number to data

featurebox.featurizers.atom.mapper.**get_atom_fea_name**(*structure: Structure*) → List[dict]

For a structure return the list of dictionary for the site occupancy for example, Fe0.5Ni0.5 site will be returned as {"Fe": 0.5, "Ni": 0.5}

Parameters
structure (*Structure*) – pymatgen Structure with potential site disorder

Returns
a list of site fraction description

featurebox.featurizers.atom.mapper.**get_atom_fea_number**(*structure: Structure*) → List[int]

Get atom features from structure, may be overwritten.

Parameters
structure – (Pymatgen.Structure) pymatgen structure

Returns
List of atomic numbers

`featurebox.featurizers.atom.mapper.get_ion_fea_name(structure: Structure)` → List[dict]

For a structure return the list of dictionary for the site occupancy for example, Fe0.5Ni0.5 site will be returned as {"Fe2+": 0.5, "Ni2+": 0.5}

Parameters

structure (*Structure*) – pymatgen Structure with potential site disorder

Returns

a list of site fraction description

`featurebox.featurizers.atom.mapper.process_atomic_orbitals(o)`

Post-processing for dict preprocessing with "1s", "2p", ...

`featurebox.featurizers.atom.mapper.process_bool_transition_metal(tm)`

Post-processing for bool preprocessing

`featurebox.featurizers.atom.mapper.process_tuple_full_electronic_structure(full_e)`

Post-processing for electronic_structure preprocessing ((1,"s",2),...)

`featurebox.featurizers.atom.mapper.process_tuple_oxidation_states(ox, size=10)`

Post-processing for tuple of float preprocessing.

`featurebox.featurizers.atom.mapper.process_uni(i)`

Post-processing for bool preprocessing General properties.

featurebox.featurizers.envir package

This part is for local environment features, mainly contains bond and state features.

Subpackages

featurebox.featurizers.envir.descriptors package

This is copy from `pyXtal_FF`. Contains atom features and bond features method.

Submodules

featurebox.featurizers.envir.desc_env module

Use descriptors form `pyXtal_FF`, in `featurebox.test_featurizers.descriptors` all with `calculate` method.

`featurebox.featurizers.envir.desc_env.get_5_result(d: Dict, **kwargs)` → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Change each center atoms has fill_size neighbors. More neighbors would be abandoned. Insufficient neighbors would be duplicated.

Parameters

- **d** – dict, dict of descriptor. at lest contain "x" and "dxdr"
- **fill_size** – int, unstable.

Returns

(center_indices,center_prop, neighbor_indices, images, distances)

center_indices: np.ndarray 1d(N,).

neighbor_indices: np.ndarray 2d(N,fill_size).

images: np.ndarray 2d(N,fill_size,l).

distance: np.ndarray 2d(N,fill_size), None. center_prop: np.ndarray 1d(N,l_c).

`featurebox.featurizers.envir.desc_env.get_strategy2_in_spheres(structure, nn_strategy, cutoff, numerical_tol=None, cutoff_name='cutoff', pbc=True)`

`featurebox.featurizers.envir.desc_env.mark_classes(classes: List)`

featurebox.featurizers.envir.environment module

`class featurebox.featurizers.envir.environment.GEONNGet(nn_strategy: str = 'UserVoronoiNN', refine: str = 'geo_refine_nn', refined_strategy_param: Optional[Dict] = None, numerical_tol=1e-08, pbc: Union[List[int], bool] = False, cutoff=5.0, check_align=True, cutoff_name='cutoff', n_jobs=1, on_errors='raise', return_type='any')`

Bases: `_BaseEnvGet`

Get properties from Pymatgen.Structure. Where the nn_strategy is from `Pymatgen`. And each structure is convert to data as following :

`center_indices`:np.ndarray of shape(n,)
center indexes.

`neighbor_indices`:np.ndarray of shape(n,fill_size)
neighbor_indexes for each center_index. *fill_size* is the parameter of *refine* function.

`images`:np.ndarray of shape(n,lb>=3)
offset vector in 3 orientations or more bond properties.

`distances`:np.ndarray of shape(n,fill_size)
distance of neighbor_indexes for each center_index.

`center_properties`:np.ndarray of shape(n,l_c)
center properties.

Parameters

- **`nn_strategy`** (*`str`*) – ["find_points_in_spheres", "find_xyz_in_spheres", "BrunnerNN_reciprocal", "BrunnerNN_real", "BrunnerNN_relative", "EconNN", "CrystalNN", "MinimumDistanceNNAll", "find_points_in_spheres","UserVoronoiNN", "ACSF","BehlerParrinello","EAD","EAMD","SOAP","SO3","SO4_Bispectrum","wACSF",]
- **`refine`** (*`str`*) – sort method for neighbors of each atom. See Also: `universe_refine_nn()`
- **`refined_strategy_param`** (*`dict`*) – parameters for refine
- **`numerical_tol`** (*`float`*) – numerical_tol

- **pbc** (*list*) – periodicity in 3 direction 3-length list,each one is 1 or 0. such as [0,0,0],The 1 mean in this direction is with periodicity.
- **cutoff** – cutoff radius

convert(*structure: Union[Structure, Molecule]*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

Parameters
structure – Structure,pymatgen Structure

Returns
center_indices,center_prop,neighbor_indices,images,distances

get_radius(*structure: Union[Structure, Molecule]*, *cutoff*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

For quick get bond distance

get_strategy1(*structure: Union[Structure, Molecule]*, *cutoff*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

For get bond distance with different strategy, for different nn_staagy could be rewrite.

get_strategy2(*structure: Union[Structure, Molecule]*, *cutoff*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

For get bond distance with different strategy, for different nn_staagy could re-write.

get_xyz(*structure: Union[Structure, Molecule]*, *cutoff*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

For quick get bond distance

featurebox.featurizers.envir.environment.**geo_refine_nn**(*center_indices*, *neighbor_indices*, *vectors*, *distances*, *center_prop=None*, *ele_numbers=None*, *fill_size=10*, *dis_sort=True*, ***kwargs*)

Change each center atoms has fill_size neighbors. More neighbors would be abandoned. Insufficient neighbors would be duplicated.

Parameters

- **center_indices** – np.ndarray 1d
- **neighbor_indices** – np.ndarray 1d
- **distances** – np.ndarray 1d or np.ndarray 2d
- **vectors** – np.ndarray 2d
- **center_prop** – np.ndarray 2d
- **ele_numbers** – np.ndarray 1d
- **fill_size** – float, not use,
- **dis_sort** – bool sort neighbors with distance.

Returns

(center_indices,center_indices, neighbor_indices, images, distances)

center_indices: np.ndarray 1d(N,).

neighbor_indices: np.ndarray 2d(N,fill_size).

images: np.ndarray 2d(N,fill_size,l).

distance: np.ndarray 2d(N,fill_size,1). center_prop: np.ndarray 1d(N,l_c).

where l, and l_c >= 1

`featurebox.featurizers.envir.environment.get_marked_class(nn_strategy, env_dict: Optional[Dict] = None, instantiation: bool = True)`

Parameters

- **nn_strategy** (*Any*) – "find_points_in_spheres", "find_xyz_in_spheres" , or nn_strategy
- **env_dict** (*dict*) – pre-definition, {"classname": class}.
- **instantiation** (*bool*) – return class of object.

Returns
object or class in NNDict.

Return type
obj

featurebox.featurizers.envir.local_env module

Use NearNeighbors from `pymatgen`. all with `get_all_nn_info` method.

Most in `pymatgen.analysis.local_env` or in `pymatgen.optimization.neighbors.find_points_in_spheres()` The costumed as following:

class `featurebox.featurizers.envir.local_env.AllAtomPairs`

Bases: `NearNeighbors`

Get all combinations of atoms as bonds in a molecule

get_nn_info(*molecule: Molecule, n: int*) → List[Dict]

Get near neighbor information :param molecule: pymatgen Molecule :type molecule: Molecule :param n: number of molecule :type n: int

Returns: List of neighbor dictionary

class `featurebox.featurizers.envir.local_env.MinimumDistanceNNAll(cutoff: float = 4.0)`

Bases: `NearNeighbors`

Determine bonded sites by fixed cutoff.

Parameters
cutoff (*float*) – cutoff radius in Angstrom to look for trial near-neighbor sites (default: 4.0).

get_nn_info(*structure: Structure, n: int*) → List[Dict]

Get all near-neighbor sites as well as the associated image locations and weights of the site with index n using the closest neighbor distance-based method.

Parameters

- **structure** (*Structure*) – input structure.
- **n** (*int*) – index of site for which to determine near neighbors.

Returns

tuples, each one of which represents a neighbor site, its image location,
and its weight.

Return type
(list of tuples (Site, array, float))

class featurebox.featurizers.envir.local_env.**UserVoronoiNN**(*tol=0, targets=None, cutoff=13.0, allow_pathological=False, weight='solid_angle', extra_nn_info=True, compute_adj_neighbors=True*)

Bases: VoronoiNN

Not for all structure.

Parameters

- **tol** (*float*) – tolerance parameter for near-neighbor finding. Faces that are smaller than *tol* fraction of the largest face are not included in the tessellation. (default: 0).
- **targets** (*Element or list of Elements*) – target element(s).
- **cutoff** (*float*) – cutoff radius in Angstrom to look for near-neighbor atoms. Defaults to 13.0.
- **allow_pathological** (*bool*) – whether to allow infinite vertices in determination of Voronoi coordination.
- **weight** (*string*) – available in get_voronoi_polyhedra)
- **extra_nn_info** (*bool*) –
- **compute_adj_neighbors** (*bool*) – adjacent. Turn off for faster performance

featurebox.featurizers.envir.local_env.**get_strategy1_in_spheres**(*structure: Union[Structure, Molecule], nn_strategy: NearNeighbors, cutoff: float = 5.0, numerical_tol: float = 1e-08, pbc=True*) → Tuple[ndarray, ndarray, ndarray, ndarray, ndarray]

featurebox.featurizers.envir.local_env.**mark_classes**(*classes: List*)

featurebox.featurizers.state package

Submodules

featurebox.featurizers.state.extrastats module

General methods for computing property statistics from a list of values

class featurebox.featurizers.state.extrastats.**PropertyStats**

Bases: object

This class contains statistical operations that are commonly employed when computing features. The primary way for interacting with this class is to call the `calc_stat` function, which takes the x_name of the statistic you would like to compute and the weights/values of datamnist to be assessed. For example, computing the mean of a list looks like:

```
>>> x = [1, 2, 3]
>>> PropertyStats.calc_stat(x, 'mean') # Result is 2
>>> PropertyStats.calc_stat(x, 'mean', weights=[0, 0, 1]) # Result is 3
```

Some the statistics functions take options (e.g., Holder means). You can pass them to the statistics functions by adding them after the x_name and two colons. For example, the 0th Holder mean would be:

>>>PropertyStats.calc_stat(x, 'holder_mean::0')

You can, of course, call the statistical functions directly. All take at least two arguments. The first is the datamnist being assessed and the second, optional, argument is the weights.

static avg_dev(*data_lst*, *weights=None*)

Mean absolute deviation of list of element datamnist. This is computed by first calculating the mean of the list, and then computing the average absolute difference between each value and the mean. :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
mean absolute deviation

static calc_stat(*data_lst*, *stat*, *weights=None*)

Compute a property statistic

Parameters

- **data_lst** (*list of floats*) – list of values
- **stat** (*str*) –
- **example** (*should be added after the x_name and separated by two colons. For*) –
- **would** (*the 2nd Holder mean*) –
- **"holder_mean::2"** (*be*) –
- **weights** (*list of floats*) – (Optional) weights for each element in data_lst

Returns
float - Desired statistic

static eigenvalues(*data_lst*, *symm=False*, *sort=False*)

Return the eigenvalues of a matrix as a numpy array :param data_lst: (matrix-like) of values :param symm: whether to assume the matrix is symmetric :param sort: wheter to sort the eigenvalues

Returns: eigenvalues

static flatten(*data_lst*, *weights=None*)

Returns a flattened copy of data_lst-as a numpy array

static geom_std_dev(*data_lst*, *weights=None*)

Geometric standard deviation :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
geometric standard deviation

static holder_mean(*data_lst*, *weights=None*, *power=1*)

Get Holder mean :param data_lst: (list/array) of values :param weights: (list/array) of weights :param power: (int/float/str) which holder mean to compute

Returns: Holder mean

static inverse_mean(*data_lst*, *weights=None*)

Mean of the inverse of each entry :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
inverse mean

static kurtosis(*data_lst*, *weights=None*)

Kurtosis of a list of datamnist :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
kurtosis

static maximum(*data_lst*, *weights=None*)

Maximum value in a list :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: (ignored)

Returns
maximum value

static mean(*data_lst*, *weights=None*)

Arithmetic mean of list :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
mean value

static minimum(*data_lst*, *weights=None*)

Minimum value in a list :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: (ignored)

Returns
minimum value

static mode(*data_lst*, *weights=None*)

Mode of a list of datamnist. If multiple elements occur equally-frequently (or same weight, if weights are provided), this function will return the minimum of those values. :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
mode

static quantile(*data_lst*, *weights=None*, *q=0.5*)

Return a specific quantile. :param weights: not used :type weights: float :param data_lst: 1D datamnist list to be used for computing, quantiles :type data_lst: list or np.ndarray :param q: The quantile, as a fraction between 0 and 1. :type q: float

Returns
(float) The computed quantile of the data_lst.

static range(*data_lst*, *weights=None*)

Range of a list :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: (ignored)

Returns
range

static skewness(*data_lst*, *weights=None*)

Skewness of a list of datamnist :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
shewness

static sorted(*data_lst*, *weights=None*)

Returns the sorted data_lst

static std_dev(*data_lst*, *weights=None*)

Standard deviation of a list of element datamnist :param data_lst: List of values to be assessed :type data_lst: list of floats :param weights: Weights for each value :type weights: list of floats

Returns
standard deviation

featurebox.featurizers.state.state_mapper module

class featurebox.featurizers.state.state_mapper.**StructurePymatgenPropMap**(*prop_name=None*, *func: Optional[Callable] = None*, *return_type='df'*, ***kwargs*)

Bases: `_StructurePymatgenPropMap`

Get property of pymatgen structure preprocessing. default ["density", "volume", "ntypesp"]

Examples

```
>>> tmps = StructurePymatgenPropMap()
>>> tmps.fit_transform()
```

Parameters

- **prop_name** – (str,list of str) prop name or list of prop name default ["density", "volume", "ntypesp"]
- **func** – (callable or list of callable) please make sure the size of it is the same with prop_name.

featurebox.featurizers.state.statistics module

class featurebox.featurizers.state.statistics.**BaseCompositionFeature**(*data_map: BinaryMap*, *n_jobs: int = 1*, *on_errors: str = 'raise'*, *return_type: str = 'df'*, *feature_labels_mark: Optional[str] = None*)

Bases: *BinaryMap*

BaseCompositionFeature is the basis for composition data. the subclass should be re-implemented, such as:

```
def mix_function(self, elems:List, nums:List):
    w_ = np.array(nums)
    return w_.dot(elems)
```

Base class for composition feature.

convert_dict(*atoms: dict*) → ndarray

Convert atom {symbol: fraction} list to numeric features

convert_number(*atoms: List*)

Convert atom {symbol: fraction} list to numeric features

fit(**args*, *x_labels=None*, ***kwargs*)

fit function in `BaseFeature` are weakened and just pass parameter.

abstract mix_function(*elems: List*, *nums: Union[List, ndarray]*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**DepartElementFeature**(*data_map: BinaryMap*, *n_composition: int*, *n_jobs: int = 1*, *on_errors: str = 'raise'*, *return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

Get the table of element data.

Examples

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> from featurebox.featurizers.state.union import UnionFeature
>>> from featurebox.featurizers.state.statistics import DepartElementFeature
>>> data_map = AtomJsonMap(search_tp="name",embedding_dict="ele_megnet.json", n_
↪jobs=1) # keep this n_jobs=1 and return_type="np"
>>> wa = DepartElementFeature(data_map,n_composition=2, n_jobs=1, return_type="pd")
>>> comp = [{"H": 2, "Pd": 1},{"He":1, "Al":4}]
>>> wa.set_feature_labels(["fea_{}".format(_) for _ in range(16)]) # 16 this the␣
↪feature number of built-in "ele_megnet.json"
>>> wa.fit_transform(comp)
   depart_fea_0_0  depart_fea_0_1  ...  depart_fea_15_0  depart_fea_15_1
0        0.352363        0.561478  ...        -0.270104        -0.212607
1       -0.067220        0.025758  ...        -0.042185         0.080350

[2 rows x 32 columns]
```

Base class for composition feature.

convert_dict(*atoms: Union[dict, Composition]*) → ndarray

Convert atom {symbol: fraction} list to numeric features

convert_number(*atoms: List*) → ndarray

Convert atom {symbol: fraction} list to numeric features

mix_function(*elems: ndarray*, *nums=None*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

set_feature_labels(*values*)

Generate attribute names.

Returns
([str]) attribute labels.

class featurebox.featurizers.state.statistics.**ExtraMix**(*data_map: BinaryMap*, *stats: Tuple[str] = ('mean',)*, *n_jobs: int = 1*, *on_errors: str = 'raise'*, *return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

See also:

WeightedSum

Base class for composition feature.

mix_function(*elems*, *nums*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**GeometricMean**(*data_map: BinaryMap*, *n_jobs: int = 1*, *on_errors: str = 'raise'*, *return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

See also:

WeightedSum

Base class for composition feature.

mix_function(*elems: ndarray*, *nums*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**HarmonicMean**(*data_map: BinaryMap, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

See also:

WeightedSum

Base class for composition feature.

mix_function(*elems, nums*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**MaxPooling**(*data_map: BinaryMap, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

See also:

WeightedSum

Base class for composition feature.

mix_function(*elems, _*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**MinPooling**(*data_map: BinaryMap, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

See also:

WeightedSum

Base class for composition feature.

mix_function(*elems*, _)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**WeightedAverage**(*data_map: BinaryMap, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

Examples

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap, AtomJsonMap
>>> data_map = AtomJsonMap(search_tp="name", n_jobs=1)
>>> wa = WeightedAverage(data_map, n_jobs=1,return_type="df")
>>> x3 = [{"H": 2, "Pd": 1},{"He":1,"Al":4}]
>>> wa.fit_transform(x3)
         0         1         2   ...        13        14        15
0  0.422068  0.360958  0.201433  ... -0.459164 -0.064783 -0.250939
1  0.007163 -0.471498 -0.072860  ...  0.206306 -0.041006  0.055843

[2 rows x 16 columns]
```

```
>>> wa.set_feature_labels(["fea_{}".format(_) for _ in range(16)])
>>> wa.fit_transform(x3)
   wt_ave_fea_0  wt_ave_fea_1  ...  wt_ave_fea_14  wt_ave_fea_15
0      0.422068      0.360958  ...      -0.064783      -0.250939
1      0.007163     -0.471498  ...      -0.041006       0.055843

[2 rows x 16 columns]
```

Base class for composition feature.

mix_function(*elems*, *nums*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**WeightedSum**(*data_map: BinaryMap, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

Examples

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap, AtomJsonMap
>>> data_map = AtomTableMap(search_tp="name", n_jobs=1)
>>> wa = WeightedSum(data_map, n_jobs=1,return_type="df")
>>> x3 = [{"H": 2, "Pd": 1},{"He":1,"Al":4}]
>>> wa.fit_transform(x3)
   wt_sum_1s  wt_sum_2s  wt_sum_2p  ...  wt_sum_6d  wt_sum_6f  wt_sum_7s
0    8320.18   11837.27      11.80  ...        0.0        0.0        0.0
1    2188.73    1513.40     986.16  ...        0.0        0.0        0.0

[2 rows x 19 columns]
```

Base class for composition feature.

mix_function(*elems, nums*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

class featurebox.featurizers.state.statistics.**WeightedVariance**(*data_map: BinaryMap, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseCompositionFeature*

See also:

WeightedSum

Base class for composition feature.

mix_function(*elems: ndarray, nums*)

Parameters

- **elems** (*list*) – Elements in compound.
- **nums** (*list*) – Number of each element.

Returns
descriptor

Return type
numpy.ndarray

featurebox.featurizers.state.union module

class featurebox.featurizers.state.union.**PolyFeature**(*, *degree: Union[int, List[int]] = 3*, *n_jobs=1*, *on_errors='raise'*, *return_type='df'*)

Bases: *BaseFeature*, ABC

Extension method.

Such as degree = 2 means (x1x2,x1**2,x2**2)

Examples

```
>>> n = np.array([[0,1,2,3,4,5],[0.422068,0.360958,0.201433,-0.459164,-0.064783,-0.
↪250939]]).T
>>> ps = pd.DataFrame(n,columns=["f1","f2"],index= ["x0","x1","x2","x3","x4","x5"])
>>> pf = PolyFeature(degree=[1,2])
>>> pf.fit_transform(n)
   f0^1      f1^1  f0^2  f0^1*f1^1      f1^2
0   0.0  0.422068   0.0   0.000000  0.178141
1   1.0  0.360958   1.0   0.360958  0.130291
2   2.0  0.201433   4.0   0.402866  0.040575
3   3.0 -0.459164   9.0  -1.377492  0.210832
4   4.0 -0.064783  16.0  -0.259132  0.004197
5   5.0 -0.250939  25.0  -1.254695  0.062970
```

Parameters

- **batch_size** (*int*) – size of batch.
- **batch_calculate** (*bool*) – batch_calculate or not.
- **n_jobs** (*int*) – Parallel number.
- **on_errors** (*str*) – How to handle the exceptions in a feature calculations. Can be nan, keep, raise. When 'nan', return a column with np.nan. The length of column corresponding to the number of feature labs. The default is 'raise' which will raise up the exception.
- **return_type** (*str*) – Specific the return type. Can be any, np,``array`` and df. 'array' and 'df' force return type to np.ndarray and pd.DataFrame respectively. If 'any', without type conversion . Default is 'any'

fit_transform(*X: Union[ndarray, DataFrame]*, *y=None*, ***kwargs*)

If *convert* takes multiple inputs, supply inputs as a list of tuples.

Copy from Mixin class for all transformers in scikit-learn. TransformerMixin

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*list*) – list of case.
- **y** (*None*) – deprecated.
- ****kwargs** – Additional fit or transform parameters. feature_labels_mark: str, mark for each feature_labes. for return_type =='pd'. x_labels: list, mark for each row. for return_type =='pd'.

Returns
result data.

Return type
X_new

set_feature_labels(*input_features=None*)

Generate attribute names.

Returns
([str]) attribute labels.

class featurebox.featurizers.state.union.**UnionFeature**(*comp: List[Dict], couple_data: Union[DataFrame, ndarray], couple=2, stats=('mean',), n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df'*)

Bases: *BaseFeature*

Transform method should input0 comp_index rather than entries.

Examples

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap, AtomJsonMap
>>> data_map = AtomJsonMap(search_tp="name", n_jobs=1)
>>> wa = DepartElementFeature(data_map,n_composition=2, n_jobs=1,return_type="df")
>>> x3 = [{"H": 2, "Pd": 1},{"He":1,"Al":4}]
>>> wa.set_feature_labels(["fea_{}".format(_) for _ in range(16)])
>>> wa.fit_transform(x3)
   depart_fea_0_0  depart_fea_0_1  ...  depart_fea_15_0  depart_fea_15_1
0        0.352363        0.561478  ...        -0.270104        -0.212607
1       -0.067220        0.025758  ...        -0.042185         0.080350

[2 rows x 32 columns]
```

```
>>> couple_data = wa.fit_transform(x3)
>>> uf = UnionFeature(x3,couple_data,couple=2,stats=("mean","maximum"))
>>> uf.fit_transform()
   mean_fea_0  maximum_fea_0  ...  mean_fea_15  maximum_fea_15
0    0.422068       0.360958  ...     0.021095       -0.212607
1    0.007163      -0.471498  ...     0.165278        0.080350

[2 rows x 32 columns]
```

```
>>> couple_data = wa.fit_transform(x3)
>>> uf = UnionFeature(x3,couple_data,couple=2,stats=("std_dev",))
>>> uf.fit_transform()
   std_dev_fea_0  std_dev_fea_1  ...  std_dev_fea_14  std_dev_fea_15
0       0.147867       0.583352  ...        0.182177        0.040657
1       0.065745       0.541477  ...        0.182331        0.086646

[2 rows x 16 columns]
```

Parameters

- **batch_size** (*int*) – size of batch.
- **batch_calculate** (*bool*) – batch_calculate or not.
- **n_jobs** (*int*) – Parallel number.
- **on_errors** (*str*) – How to handle the exceptions in a feature calculations. Can be nan, keep, raise. When 'nan', return a column with np.nan. The length of column corresponding to the number of feature labs. The default is 'raise' which will raise up the exception.
- **return_type** (*str*) – Specific the return type. Can be any, np,``array`` and df. 'array' and 'df' force return type to np.ndarray and pd.DataFrame respectively. If 'any', without type conversion . Default is 'any'

convert(*comp_number=0*)

Get elemental property attributes

Parameters
comp – Pymatgen composition object

Returns
Specified property statistics of features :param comp_number:

Return type
all_attributes

fit_transform(*entries: Optional[List] = None*) → Any

If *convert* takes multiple inputs, supply inputs as a list of tuples.

Copy from Mixin class for all transformers in scikit-learn. TransformerMixin

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*list*) – list of case.
- **y** (*None*) – deprecated.
- ****kwargs** – Additional fit or transform parameters. feature_labels_mark: str, mark for each feature_labes. for return_type =='pd'. x_labels: list, mark for each row. for return_type =='pd'.

Returns
result data.

Return type
X_new

set_feature_labels(*self_elem_data_columns_values: List*)

Generate attribute names.

Parameters
self_elem_data_columns_values (*List*) – name

Return type
([str]) attribute labels.

transform(*entries: Optional[List] = None*) → Any

Transform a list of entries. Each iterable element of entries is corresponding to the parameter of `convert`, If `convert` takes n multiple inputs, the transform inputs should be a list or tuple (size n),

[(p1,p2),(p1,p2),(p1,p2),...,(p1,p2),(p1,p2)]

which can be from *zip* ` or used the built-in `transform_with_zip`.

Parameters
entries (*list*) – A list of entries to be featured.

Returns
result – features for each entry.

Return type
any

Submodules

featurebox.featurizers.base_feature module

Base

class featurebox.featurizers.base_feature.**BaseFeature**(*n_jobs: int = 1, *, on_errors: str = 'raise', return_type: str = 'any', batch_calculate: bool = False, batch_size: int = 30, feature_labels_mark: Optional[str] = None, **kwargs*)

Bases: `object`

Using a BaseFeature Class

That means you can embed this feature directly into `BaseFeature` class implement.

```
class MatFeature(BaseFeature):
    def convert(spath, *x):
        ...
```

`BaseFeature` implement `sklearn.base.BaseEstimator` and `sklearn.base.TransformerMixin` that means you can use it in a scikit-learn way.

```
feature = SomeFeature()
features = feature.fit_transform(X)
```

Note: The `convert` method should be rewrite to deal with single case. And the `transform` and `fit_transform` will be established for list of case automatically.

Adding references

`BaseFeature` also provide you to retrieving proper references for a feature. The `__citations__` returns a list of papers that should be cited. The `__authors__` returns a list of people who wrote the feature. Also can be accessed from property `citations` and `citations`.

These operations must be implemented for each new feature:

- `feature_labels` - Generates a human-meaningful x_name for each of the features. Implement this as property.

which can be set by `set_feature_labels`

Also suggest to implement these two properties:

- `citations` - Returns a list of citations in BibTeX format.
- `authors` - Returns a list of people who contributed writing a paper.

Note: None of these operations should change the state of the feature. I.e., running each method twice should no produce different results, no class attributes should be changed, Running one operation should not affect the output of another.

Parameters

- **batch_size** (*int*) – size of batch.
- **batch_calculate** (*bool*) – batch_calculate or not.
- **n_jobs** (*int*) – Parallel number.
- **on_errors** (*str*) – How to handle the exceptions in a feature calculations. Can be `nan`, `keep`, `raise`. When 'nan', return a column with np.nan. The length of column corresponding to the number of feature labs. The default is 'raise' which will raise up the exception.
- **return_type** (*str*) – Specific the return type. Can be `any`, `np`,``array`` and `df`. 'array' and 'df' force return type to np.ndarray and pd.DataFrame respectively. If 'any', without type conversion . Default is 'any'

property authors

List of implementors of the feature.

Returns

(list) each element should either be a string with author x_name (e.g., "Anubhav Jain") or a dictionary with required key "x_name" and other keys like "email" or "institution" (e.g., {"x_name": "Anubhav Jain", "email": "ajain@lbl.gov", "institution": "LBNL"}).

property citations

Citation(s) and reference(s) for this feature.

Returns

(list) each element should be a string citation, ideally in BibTeX format.

convert(*d*)

Main feature function, which has to be implemented in any derived feature subclass.

Notes

It cannot be passed np.ndarray in default unless:

1. useful for bond_converter. For np.array we check the ndim and for ndim 2, or 3. we decide whether to pass them the data to `_converter` together or separately by `self.ndim` attribute. Now max support 3d. due to for some functions, using `ufunc` in numpy is very efficient.

2. keep the size of data and simple the `_convert`.

Parameters
d – one input data (one sample, one case),

Returns
new x.

Return type
new_x

static emptytonone(*d*)

property feature_labels

Generate attribute names.

Returns
([str]) attribute labels.

fit(**args*, ***kwargs*)

fit function in `BaseFeature` are weakened and just pass parameter.

fit_transform(*X: List*, *y=None*, ***kwargs*) → Any

If *convert* takes multiple inputs, supply inputs as a list of tuples.

Copy from Mixin class for all transformers in scikit-learn. TransformerMixin

Fit to data, then transform it.

Fits transformer to *X* and *y* with optional parameters *fit_params* and returns a transformed version of *X*.

Parameters

- **X** (*list*) – list of case.
- **y** (*None*) – deprecated.
- ****kwargs** – Additional fit or transform parameters. feature_labels_mark: str, mark for each feature_labes. for return_type =='pd'. x_labels: list, mark for each row. for return_type =='pd'.

Returns
result data.

Return type
X_new

property n_jobs

int Parallel number.

Type
n_jobs

static nonetoempty(*d*)

set_feature_labels(*values: List[str]*)

Generate attribute names.

Returns
([str]) attribute labels.

transform(*entries: List*) → Any

Transform a list of entries. Each iterable element of entries is corresponding to the parameter of `convert`, If `convert` takes n multiple inputs, the transform inputs should be a list or tuple (size n),

[(p1,p2),(p1,p2),(p1,p2),...,(p1,p2),(p1,p2)]

which can be from *zip* ` or used the built-in `transform_with_zip`.

Parameters
entries (*list*) – A list of entries to be featured.

Returns
result – features for each entry.

Return type
any

transform_with_zip(**args*) → Any

Second transform, which convert Iterables to list and run transform.

first: p1s,p2s -> [(p1,p2),(p1,p2),(p1,p2),...,(p1,p2),(p1,p2)]

second: run self.transform

Parameters
args (*Iterable*) – each of args must be Iterable.

Returns
result – features for each entry.

Return type
any

featurebox.featurizers.base_feature.**Converter**

alias of *BaseFeature*

class featurebox.featurizers.base_feature.**ConverterCat**(**args:* BaseFeature, *force_concatenate=False*, *n_jobs: int = 1*, *on_errors: str = 'raise'*, *return_type: str = 'any'*)

Bases: *BaseFeature*

Pack the converters in to one unified approach. The same type Converter would merge and different would order to run. Thus, keeping the same type is next to each other! such as A(),A(),B(),B().

Examples

```
>>> tmps = ConverterCat(
...     AtomEmbeddingMap(),
...     AtomEmbeddingMap("ie.json")
...     AtomTableMap(search_tp="name"))
>>> tmp.convert(x)
>>> tmp.tranmform(xs)
```

Parameters
args (*Converter*) – List of Converter

convert(*d*)

convert and concatenate.

static sums(*args*)

SUM

class featurebox.featurizers.base_feature.**ConverterSequence**(**args: BaseFeature, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'any'*)

Bases: *BaseFeature*

Pack the converters in to one sequentially executed assembly approach.

input -> convert1 -> temp -> convert2 -> temp -> convert3 -> output

Notes

There is no error checking, please make sure the `temp` could be passed manually !!! There is no error checking, please make sure the `temp` could be passed manually !!! There is no error checking, please make sure the `temp` could be passed manually !!!

Examples

```
>>> tmps = ConverterCat(
...     AtomEmbeddingMap(),
...     DummyConverter()
>>> tmp.convert(x)
```

Parameters
args (*Converter*) – List of Converter

convert(*d*)

convert batched

class featurebox.featurizers.base_feature.**DummyConverter**(*n_jobs: int = 1, *, on_errors: str = 'raise', return_type: str = 'any', batch_calculate: bool = False, batch_size: int = 30, feature_labels_mark: Optional[str] = None, **kwargs*)

Bases: *BaseFeature*

Dummy converter as a placeholder, Do nothing.

Parameters

- **batch_size** (*int*) – size of batch.
- **batch_calculate** (*bool*) – batch_calculate or not.
- **n_jobs** (*int*) – Parallel number.
- **on_errors** (*str*) – How to handle the exceptions in a feature calculations. Can be nan, keep, raise. When 'nan', return a column with np.nan. The length of column corresponding to the number of feature labs. The default is 'raise' which will raise up the exception.
- **return_type** (*str*) – Specific the return type. Can be any, np,``array`` and df. 'array' and 'df' force return type to np.ndarray and pd.DataFrame respectively. If 'any', without type conversion . Default is 'any'

convert(*d*) → ndarray

Dummy convert, does nothing to input.

Parameters

d (*Any*) – input object

Returns: d

featurebox.featurizers.batch_feature module

class featurebox.featurizers.batch_feature.**BatchFeature**(*data_type: str = 'compositions', user_convert: Optional[BaseFeature] = None, n_jobs: int = 1, on_errors: str = 'raise', return_type: str = 'df', batch_calculate: bool = False, batch_size: int = 30*)

Bases: object

Script for generate batch_data, could be copied and user-defined.

Parameters

- **data_type** (*str*) – Predefined name ["elements", "compositions", "structures"]
- **user_convert** (*BatchFeature*) – which contain *convert* method.

convert(*d*)

property feature_labels

fit_transform(*entries: List*)

set_feature_labels(*values: List[str]*)

transform(*entries: List*)

featurebox.selection package

Submodules

featurebox.selection.backforward module

Forward_and_back feature elimination for feature ranking

class `featurebox.selection.backforward.`**`BackForward`**(*estimator: BaseEstimator, n_type_feature_to_select: Optional[int] = None, primary_feature: Optional[int] = None, multi_grade: int = 2, multi_index: Optional[List] = None, refit=True, cv=5, min_type_feature_to_select: int = 3, must_index: Optional[List] = None, tolerant: float = 0.01, verbose: int = 1, random_state: Optional[int] = None, scoring: Optional[str] = None, note: bool = True, filter_warn: bool = False*)

Bases: `BaseEstimator`, `MetaEstimatorMixin`, `SelectorMixin`, *`MultiBase`*

BackForward method to selected features.

`n_feature_`

The number of selected features finally.

Type
int

`support_`

The mask of selected features finally.

Type
array of shape [n_feature]

`estimator_`

The best model with the best features finally (refited with all data.).

Type
object

`best_score_`

Best score of best model of best features.

Type
float

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> svr= SVR()
>>> bf = BackForward(svr,primary_feature=4, random_state=1,verbose=0,note=False)
```

(continues on next page)

(continued from previous page)

```
>>> new_x = bf.fit_transform(X,y)
>>> bf.support_
array([False,  True,  True, False, False, False, False,  True])
```

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> from sklearn.model_selection import cross_val_score
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> X_train,y_train,X_test,y_test = X[:50],y[:50],X[-50:],y[-50:]
```

```
>>> svr= SVR()
>>> bf = BackForward(svr, primary_feature=4, random_state=1, refit=True, cv=5,
↪verbose=0,note=False)
>>> bf = bf.fit(X_train,y_train)
>>> bf.best_score_          # cv score
-3.0552830696940037
>>> train_score = bf.score(X_train,y_train)  # train score
>>> test_score = bf.score(X_test,y_test) # test score in more data.
>>> np.mean(cross_val_score(bf.estimator_,X_train[:,bf.support_],y_train,cv=5)) #␣
↪get cv_score manually.
-3.0552830696940037
```

Notes

If `score` and `predict` is used, the `refit` should be set True, the refit used all data in `fit` function, that is, it is not test score/predict.

Examples

If GridSearchCV, the refit should be set True and return the cv score.

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> from sklearn import model_selection
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> X_train,y_train,X_test,y_test = X[:50],y[:50],X[-50:],y[-50:]
```

```
>>> svr= SVR()
>>> gd = model_selection.GridSearchCV(svr,param_grid={"C":[1,10]},n_jobs=1)  # keep␣
↪n_jobs=1 there.
>>> bf = BackForward(gd,primary_feature=4, random_state=1, refit=True,
... scoring="neg_root_mean_squared_error",cv=5,verbose=0, note=False)
Uniform parameter in SearchCV and Exhaustion:
(scoring=neg_root_mean_squared_error, cv=5, refit=True)
```

(continues on next page)

(continued from previous page)

```
>>> bf = bf.fit(X_train,y_train)
>>> bf.best_score_          # cv score
-0.5919173121895709
```

```
>>> train_score = bf.score(X_train,y_train)  # train score
>>> test_score = bf.score(X_test,y_test) # test score in more data.
>>> # bf.estimator_ is the gd object (GridSearchCV)
>>> bf.estimator_.best_score_ # re cv_score in manually.
-0.5919173121895709
```

Parameters

- **estimator** (*estimator object*) – This is assumed to implement the scikit-learn estimator interface. A supervised sklearn learning estimator with `fit` method.
- **n_type_feature_to_select** (*int*) – The max number of feature to selection. If `None`, select the features with best score.
- **min_type_feature_to_select** (*int*) – force select number min.
- **primary_feature** (*int*) – primary features to start loop, default initial n_features//2.
- **multi_grade** (*int*) – group number.
- **multi_index** – group index.
- **must_index** – must selection index.
- **tolerant** – tolerant for rank compare.
- **verbose** (*int*) – print or not.
- **random_state** (*int*) – random_state.
- **refit** (*bool*) – refit or not. if refit, the model would use all data.
- **scoring** (*None,str*) – scoring method name.
- **note** (*bool*) – print note or not.
- **filter_warn** (*bool*) – warnings.filterwarnings or not.

fit(*X*, *y*)

Fit the baf model and then the underlying estimator on the selected
feature.

Parameters

- **X** (*{array-like, sparse matrix}, shape = [n_samples, n_feature]*) – The training input0 samples.
- **y** (*array-like, shape = [n_samples]*) – The target values.

predict(*X*)

Reduce X to the selected feature and then using the underlying estimator to predict.
Only available `refit=True`.

Parameters
X (*array of shape [n_samples, n_feature]*) – The input0 samples.

Returns
y – The predicted target values.

Return type
array of shape [n_samples]

score(*X*, *y*, *scoring=None*)

Reduce X to the selected feature and then return the score of the underlying estimator. Only available `refit=True`.

Parameters

- **X** (*array of shape [n_samples, n_feature]*) – The input0 samples.
- **y** (*array of shape [n_samples]*) – The target values.
- **scoring** (*str, callable, default=None*) – Strategy to evaluate the performance of the cross-validated model on the test set.

  If *scoring* represents a single score, one can use: a single string (see scoring_parameter)

  The score defined by `scoring` if provided, and the `estimator_.score` method otherwise else raise error.

class `featurebox.selection.backforward.`**BackForwardStable**(*estimator: BaseEstimator, n_type_feature_to_select: Optional[int] = None, min_type_feature_to_select: int = 3, primary_feature: Optional[int] = None, multi_grade: int = 2, multi_index: Optional[List] = None, must_index: Optional[List] = None, verbose: int = 0, random_state: Optional[int] = None, tolerant: float = 0.001, cv: int = 5, times: int = 5, scoring: Optional[str] = None, n_jobs: Optional[int] = None, refit=False, note=True*)

Bases: `MetaEstimatorMixin`, `SelectorMixin`, `BaseEstimator`

BackForwardStable. Run with different order for more Stable (Just for test).

n_feature_

The number of selected feature with cross-validation.

Type
int

support_

The mask of selected feature.

Type
array of shape [n_feature]

estimator_

The model with the best features finally (refited with all data.).

Type
object

best_score_

Best score of best model of best features.

Type
float

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> svr= SVR()
>>> bf = BackForwardStable(svr,primary_feature=3, random_state=1,verbose=0,
↪note=False)
>>> new_x = bf.fit_transform(X,y)
>>> bf.support_
array([False,  True, False, False, False,  True,  True, False])
>>> bf.best_score_
-0.09122826477472024
```

If score and predict is used, the refit could be set True and make sure the data is splited, due to the refit used all data in fit() function.

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> svr= SVR()
>>> bf = BackForwardStable(svr,primary_feature=4, random_state=1, refit=True,
↪verbose=0,note=False)
>>> new_x = bf.fit_transform(X[:50],y[:50])
>>> train_score = bf.score(X[50:],y[50:])
>>> cv_score = bf.best_score_
...
```

If GridSearchCV, the refit could be set True and return the cv score.

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> from sklearn import model_selection
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> svr= SVR()
>>> gd = model_selection.GridSearchCV(svr,param_grid={"C":[1,10]})
>>> bf = BackForward(gd,primary_feature=4, random_state=1, refit=True, cv=5,
↪verbose=0,note=False)
Uniform parameter in SearchCV and Exhaustion:
(scoring=None, cv=5, refit=True)
```

(continues on next page)

(continued from previous page)

```
>>> new_x = bf.fit_transform(X,y)
...
```

Parameters

- **estimator** (*estimator object*) – This is assumed to implement the scikit-learn estimator interface. A supervised sklearn learning estimator with `fit` method.
- **n_type_feature_to_select** (*int*) – The max number of feature to selection. If `None`, select the features with best score.
- **min_type_feature_to_select** (*int*) – force select number min.
- **primary_feature** (*int*) – primary features to start loop, default initial n_features//2.
- **multi_grade** (*int*) – group number.
- **multi_index** – group index.
- **must_index** – must selection index.
- **tolerant** – tolerant for rank compare.
- **verbose** (*int*) – print or not.
- **random_state** (*int*) – random_state.
- **refit** (*bool*) – refit or not. if refit, the model would use all data.
- **n_jobs** (*int or None*) – Number of cores to run in parallel while fitting across folds. `None` means 1 and `-1` means using all processors.
- **scoring** (*None,str*) – scoring method.
- **note** (*bool*) – print note or not.

fit(*X*, *y*, *groups=None*)

Fit the baf model and automatically tune the number of selected feature.

Parameters

- **X** (*{array-like, sparse matrix}, shape = [n_samples, n_feature]*) – Training vector, where *n_samples* is the number of samples and *n_feature* is the total number of feature.
- **y** (*array-like, shape = [n_samples]*) – Target values (integers for classification, real numbers for regression).
- **groups** (*array-like, shape = [n_samples], optional*) – cal_group labels for the samples used while splitting the dataset into train/test set.

predict(*X*)

Reduce X to the selected feature and then Fit using the underlying estimator, only with refit. Only available `refit=True`.

Parameters

X (*array of shape [n_samples, n_feature]*) – The input0 samples.

Returns

y – The predicted target values.

Return type

array of shape [n_samples]

score(*X*, *y*, *scoring=None*)

Reduce X to the selected feature and then return the score of the underlying estimator, only with refit. Only available `refit=True`.

Parameters

- **X** (*array of shape [n_samples, n_feature]*) – The input0 samples.
- **y** (*array of shape [n_samples]*) – The target values.

featurebox.selection.corr module

Calculate the correction of columns.

class `featurebox.selection.corr.`**Corr**(*threshold: float = 0.85*, *multi_grade: int = 2*, *multi_index: Optional[List] = None*, *must_index: Optional[List] = None*, *random_state: int = 0*)

Bases: `BaseEstimator`, `MetaEstimatorMixin`, `SelectorMixin`, *`MultiBase`*

Calculate correlation. (Where the result are changed with random state.)

1. Used for filter automatically by machine

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from featurebox.selection.corr import Corr
>>> x, y = fetch_california_housing(return_X_y=True)
>>> x = x[:100]
>>> y = y[:100]
>>> co = Corr(threshold=0.5)
>>> new_x = co.fit_transform(x)
>>> select_feature = co.support_
```

1. Used for get group exceeding the threshold by setp

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from featurebox.selection.corr import Corr
>>> x, y = fetch_california_housing(return_X_y=True)
>>> x = x[:100]
>>> y = y[:100]
>>> co = Corr(threshold=0.5)
>>> groups = co.count_cof(np.corrcoef(x[:,:7], rowvar=False))
>>> groups[1]
[[0, 6], [1], [2], [3], [4], [5], [0, 6]]
>>> groups[0]
[[1.0, 0.554], [1.0], [1.0], [1.0], [1.0], [1.0], [0.554, 1.0]]
>>> co.remove_coef(groups[1]) # Filter automatically by machine.
[0, 1, 2, 3, 4, 5]
```

Where the remove_coef are changed with random state.

Where the (0,6) are with correlation more than 0.7.

3. Used for binding correlation

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from featurebox.selection.corr import Corr
>>> x, y = fetch_california_housing(return_X_y=True)
>>> x = x[:100]
>>> y = y[:100]
>>> co = Corr(threshold=0.3,multi_index=[0,8],multi_grade=2)
>>> # in range [0,8], the features are binding in to 2 sized: [[0,1],[2,3],[4,5],[6,
↪7]]
>>> co.fit(x)
Corr(multi_index=(0, 8), threshold=0.3)
```

Parameters

- **threshold** (*float*) – ranking threshold.
- **multi_grade** – binding_group size, calculate the correction between binding.
- **multi_index** (*list*) – the range of multi_grade:[min,max).
- **must_index** (*list*) – the columns force to index.
- **random_state** (*int*) –

count_cof(*cof=None*)

Check cof and count the number.

static cov_y(*x_, y_*)

filter()

fit(*data, pre_cal=None, method='mean'*)

remove_by_y(*y_*)

remove_coef(*cof_list_all*)

Delete the index of feature with repeat coef.

featurebox.selection.exhaustion module

class featurebox.selection.exhaustion.**Exhaustion**(*estimator: BaseEstimator, n_select: Tuple = (2, 3, 4), multi_grade: Optional[int] = None, multi_index: Optional[List] = None, must_index: Optional[List] = None, n_jobs: int = 1, refit: bool = False, cv: int = 5, scoring: Optional[str] = None, note=True, filter_warn=False*)

Bases: BaseEstimator, MetaEstimatorMixin, SelectorMixin, *MultiBase*

Exhaustion features combination.

n_feature_

The number of selected features finally.

Type
int

support_

The mask of selected features finally.

Type
array of shape [n_feature]

estimator_

The best model with the best features finally (refited with all data.).

Type
object

best_score_

Best score of best model of best features.

Type
float

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.model_selection import cross_val_predict
>>> from sklearn.svm import SVR
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> X_train,y_train,X_test,y_test = X[:50],y[:50],X[-50:],y[-50:]
```

```
>>> svr = SVR()
>>> bf = Exhaustion(svr,n_select=(2,),refit=True,note=False)
>>> new_x = bf.fit_transform(X,y)
>>> bf.support_
array([False, False, False,  True, False,  True, False, False])
>>> train_score = bf.score(X_train,y_train)  # train score
>>> test_score = bf.score(X_test,y_test) # test score in more data.
>>> np.mean(cross_val_score(bf.estimator_,X_train[:,bf.support_],y_train,cv=5)) #␣
↪re cv_score in manually.
-2.888471220974372
>>> np.mean(cross_val_predict(bf.estimator_,X_train[:,bf.support_],y_train,cv=5)) #␣
↪re cv_predict for plot.
1.6001222987265382
```

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> from sklearn import model_selection
>>> X,y = fetch_california_housing(return_X_y=True)
>>> X = X[:100]
>>> y = y[:100]
>>> svr= SVR()
```

```
>>> gd = model_selection.GridSearchCV(svr, param_grid=[{"C": [1, 10]}], n_jobs=1,␣
↪cv=3)
>>> bf = Exhaustion(gd,n_select=(2,),refit=True,note=False,cv=5)
Uniform parameter in SearchCV and Exhaustion:
(scoring=None, cv=5, refit=True)
>>> new_x = bf.fit_transform(X,y)
>>> bf.support_
array([False, False, False,  True, False,  True, False, False])
>>> bf.best_score_
-0.7336740728050252
```

Parameters

- **estimator** – sklearn model or GridSearchCV.
- **n_select** (*tuple*) – the n_select list,default,n_select=(3, 4).
- **multi_grade** (*list*) – binding_group size, calculate the correction between binding.
- **multi_index** (*list*) – the range of multi_grade:[min,max).
- **must_index** (*list*) – the columns force to index.
- **n_jobs** (*int*) – n_jobs.
- **refit** (*bool*) – refit or not, if refit the model would use all data.
- **cv** (*bool*) – if estimator is sklearn model, used cv, else pass.
- **scoring** (*None,str*) – scoring method name.
- **note** (*bool*) – print note or not.
- **filter_warn** (*bool*) – warnings.filterwarnings or not.

fit(*X*, *y*)

Fit the baf model and then the underlying estimator on the selected feature.

Parameters

- **X** (*{array-like, sparse matrix}, shape = [n_samples, n_feature]*) – The training input0 samples.
- **y** (*array-like, shape = [n_samples]*) – The target values.

predict(*X*)

Reduce X to the selected feature and then Fit using the underlying estimator. Only available `refit=True`.

Parameters

X (*array of shape [n_samples, n_feature]*) – The input0 samples.

Returns
y – The predicted target values.

Return type
array of shape [n_samples]

score(*X*, *y*, *scoring=None*)

Reduce X to the selected feature and then return the score of the underlying estimator. Only available `refit=True`.

Parameters

- **X** (*array of shape [n_samples, n_feature]*) – The input0 samples.
- **y** (*array of shape [n_samples]*) – The target values.
- **scoring** (*str, callable, default=None*) – Strategy to evaluate the performance of the cross-validated model on the test set.

  If *scoring* represents a single score, one can use: a single string (see scoring_parameter)

  The score defined by `scoring` if provided, and the `estimator_.score` method otherwise else raise error.

`featurebox.selection.exhaustion.`**ExhaustionCV**

alias of *Exhaustion*

featurebox.selection.ga module

class `featurebox.selection.ga.`**GA**(*estimator*, *n_jobs=2*, *pop_n=1000*, *hof_n=1*, *cxpb=0.6*, *mutpb=0.3*, *ngen=40*, *max_or_min='max'*, *mut_indpb=0.05*, *max_=None*, *min_=2*, *random_state=None*, *multi_grade=2*, *multi_index=None*, *must_index=None*, *cv: int = 5*, *scoring=None*, *filter_warn=False*)

Bases: `BaseEstimator`, `MetaEstimatorMixin`, `SelectorMixin`, *MultiBase*

GA with binding. Please just passing training data.

Examples

```
>>> from sklearn.datasets import fetch_california_housing
>>> from sklearn.svm import SVR
>>> data = fetch_california_housing()
>>> X = data.data
>>> y = data.target
>>> X_train,y_train,X_test,y_test = X[:50],y[:50],X[-50:],y[-50:]
>>> svr = SVR(gamma="scale", C=100)
>>> ga = GA(estimator=svr, n_jobs=2, pop_n=50, hof_n=1, cxpb=0.8, mutpb=0.4, ngen=3,
... max_or_min="max", mut_indpb=0.1, min_=2, multi_index=[0, 5],random_state=0)
>>> ga.fit(X_train, y_train)
gen nevals  min     max
1   50      -4.9231 -1.09124
2   43      -3.83152  -1.09124
3   46      -4.9231   -1.09124
[1, 1, 1, 1, 0, 0, 1, 0] (-1.039237326973499,)
GA(cxpb=0.8, estimator=SVR(C=100), multi_index=(0, 5), mut_indpb=0.1, mutpb=0.4,
```

(continues on next page)

(continued from previous page)

```
   ngen=3, pop_n=50, random_state=0)
>>> ga.score(X_test, y_test)
-28.542309712899435
```

Parameters

- **estimator** – sklearn estimator
- **n_jobs** (*int*) – njobs
- **pop_n** (*int*) – population
- **hof_n** (*int*) – hof
- **cxpb** (*float*) – probility of cross
- **mutpb** (*float*) – probility of mutate
- **ngen** (*int*) – generation
- **max_or_min** (*str*) – "max","min";max problem or min
- **mut_indpb** (*float*) – probility of mutate of each node.
- **max** (*int*) – max size
- **min** (*int*) – min size
- **random_state** (*float*) – randomstate
- **multi_grade** – binding grade
- **multi_index** – binding range [min,max]
- **scoring** (*None,str*) – scoring method name.
- **cv** (*bool*) – if estimator is sklearn model, used cv, else pass.
- **filter_warn** (*bool*) – warnings.filterwarnings or not.

feature_fold_length(*feature*)

fit(*X*, *y*)

Fit data and run GA.

fitness_func(*ind*, *model*, *x*, *y*, *return_model=False*)

static generate_min_max(*space*, *min_=2*, *max_=None*)

predict(*X*)

Reduce X to the selected feature and then return the score of the underlying estimator.

Parameters

X (*array of shape [n_samples, n_feature]*) – The input0 samples.

predict_func(*ind*, *model*, *x*)

score(*X*, *y*)

Reduce X to the selected feature and then return the score of the underlying estimator.

Parameters

- **X** (*array of shape [n_samples, n_feature]*) – The input0 samples.

- **y** (*array of shape [n_samples]*) – The target values.

score_cv(*X*, *y*)

Reduce X to the selected feature and then return the score of the underlying estimator.

Parameters

- **X** (*array of shape [n_samples, n_feature]*) – The input0 samples.
- **y** (*array of shape [n_samples]*) – The target values.

socre_func(*ind*, *model*, *x*, *y*, *scoring=None*)

unfold(*ind*)

featurebox.selection.ga.**eaSimple**(*population*, *toolbox*, *cxpb*, *mutpb*, *ngen*, *stats=None*, *n_jobs=2*, *halloffame=None*, *verbose=True*)

This algorithm reproduce the simplest evolutionary algorithm.

Parameters

- **population** – A list of individuals.
- **n_jobs** – jobs.
- **toolbox** – A `Toolbox` that contains the evolution operators.
- **cxpb** – The probability of mating two individuals.
- **mutpb** – The probability of mutating an individual.
- **ngen** – The number of generation.
- **stats** – A `Statistics` object that is updated inplace, optional.
- **halloffame** – A `HallOfFame` object that will contain the best individuals, optional.
- **verbose** – Whether to log the statistics.

Returns

The final population

Returns

A class:~*deap.tools.Logbook* with the statistics of the evolution

featurebox.selection.ga.**filt**(*ind*, *min_=2*, *max_=None*)

featurebox.selection.ga.**generate**(*space*)

featurebox.selection.ga.**generate_xi**()

featurebox.selection.multibase module

class featurebox.selection.multibase.**MultiBase**(*multi_grade: int = 2*, *multi_index: Optional[Union[List, Tuple]] = None*, *must_index: Optional[Union[List, Tuple]] = None*)

Bases: `object`

Base method for binding

Parameters

- **multi_grade** (*int*) – binding_group size, calculate the correction between binding

- **multi_index** (*list*, *tuple*, *None*) – the range of multi_grade:[min,max)
- **must_index** (*list*, *tuple*, *None*) – the columns force to index

property check_multi

property check_must

feature_fold(*feature*)

feature_unfold(*feature*)

inverse_transform_index(*index*)

inverse the selected index to origin index by support.

property must_fold_add

property must_unfold_add

transform(*data: Any*)

transform_index(*index*)

Get support index.

featurebox.utils package

Submodules

featurebox.utils.general module

featurebox.utils.predefined_typing module

Define several types for convenient use

featurebox.utils.quickmethod module

This is script, copy for using, rather than call.**

featurebox.utils.quickmethod.**cv_predict**(*x*, *y*, *s_estimator*, *kf*)

featurebox.utils.quickmethod.**dict_me**(*me='clf'*)

featurebox.utils.quickmethod.**dict_method_clf**()

many clf method.

featurebox.utils.quickmethod.**dict_method_reg**()

many reg method.

featurebox.utils.quickmethod.**method_pack**(*method_all*, *me='reg'*, *scoring=None*, *gd=True*, *cv=10*)

return cv or gd.

featurebox.utils.quickmethod.**pack_score**(*y_test_true_all*, *y_test_predict_all*, *scoring*)

CHAPTER

FIVE

EXAMPLES

Before reading this part, please make sure the you have already known `pymatgen.core.Structure` . *Sample Data and Background*

5.1 Batch Transform Data

If you don't have a preference or idea for features, just try with `BatchFeature` , We using features from pymatgen firstly.

- Transform structure list.

```
>>> from featurebox.featurizers.batch_feature import BatchFeature
>>> bf = BatchFeature(data_type="structures", return_type="df")
>>> data = bf.fit_transform(structure_list)
```

`structures_list` is list of `struceture` of `pymatgen`.

- Transform composition list.

```
>>> from featurebox.featurizers.batch_feature import BatchFeature
>>> bf = BatchFeature(data_type="compositions")
>>> com = [[{str(i.symbol): 1} for i in structurei.species]  for structurei in structure_
↪list]
>>> #where com is element list
>>> data = bf.fit_transform(com)
```

- Transform element list.

```
>>> from featurebox.featurizers.batch_feature import BatchFeature
>>> bf = BatchFeature(data_type="elements")
>>> aas = [[{str(i.symbol): 1} for i in structurei.species]  for structurei in structure_
↪list]
>>> data = bf.fit_transform(aas)
>>> bf.element_c.search_tp="number"
>>> aas = [[i.specie.Z for i in structure] for structure in structure_list]
>>> data = bf.fit_transform(aas)
```

Note

```
It is highly recommended that using this function as a beginner,
Because we can customize more and more powerful converters.
```

Now, try it !

5.2 Json Data

The key json must be element name, such as {"H": ... ,"He": ... }, and the structures is pymatgen `Structure` list.

- Index by structure.

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="number", embedding_dict="ele_megnet.json")
>>> a = tmps.convert(structurei)
```

The return data are properties of 1, 76 elements.

- Index by number, with your-self json.

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="number",embedding_dict="ele_megnet.json")
>>> s = [1,76]
>>> a = tmps.convert(s)
```

The return data are properties of 1, 76 elements.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0.35236	0.63595	0.21734	-0.19196	0.25375	-0.42326	0.22130	-0.45241	-1.00771	-0.28994	0.12682	-0.02593	0.71751	-0.63199	0.02110	-0.27010
1	0.33976	0.12031	-0.08316	-0.94635	-0.14058	-0.06285	0.61304	0.41731	-0.75519	0.57838	0.45582	0.08221	0.44677	0.13261	-0.76529	-0.26538

- Index by dict data.

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="name")
>>> s = [{"H": 2, }, {"Al": 1}] # or [{i.element.symbol:1} for i in structure.species]
>>> a = tmps.convert(s)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0.70473	1.27190	0.43468	-0.38391	0.50750	-0.84652	0.44259	-0.90482	-2.01543	-0.57987	0.25364	-0.05186	1.43502	-1.26399	0.04219	-0.54021
1	0.02576	-0.62465	-0.13220	-0.16448	-0.09625	0.11413	0.13666	0.06773	-0.09260	0.20645	0.37388	-0.17317	-0.11670	0.31218	-0.09258	0.08035

- Batch data.

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="name")
>>> s = [[{"H": 2, }, {"Ce": 1}],[{"H": 2, }, {"Al": 1}]]
>>> a = tmps.transform(s)
```

The return data are list of 2 np.ndarray data.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0.70473	1.27190	0.43468	-0.38391	0.50750	-0.84652	0.44259	-0.90482	-2.01543	-0.57987	0.25364	-0.05186	1.43502	-1.26399	0.04219	-0.54021
1	-0.19783	0.14517	-0.43861	0.11368	0.04111	0.75552	0.01807	0.40857	0.16970	0.13481	-0.07081	-0.08759	-0.33948	0.20795	0.02751	0.23956

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0.70473	1.27190	0.43468	-0.38391	0.50750	-0.84652	0.44259	-0.90482	-2.01543	-0.57987	0.25364	-0.05186	1.43502	-1.26399	0.04219	-0.54021
1	0.02576	-0.62465	-0.13220	-0.16448	-0.09625	0.11413	0.13666	0.06773	-0.09260	0.20645	0.37388	-0.17317	-0.11670	0.31218	-0.09258	0.08035

5.3 Table data

Read table data as following format, and organize by composition.

Data	F0	F1	...
H	V	V	...
He	V	V	...
Li	V	V	...
Be	V	V	...
...	...	...	...

- Then run the code.

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap
>>> tmps = AtomTableMap(search_tp="name",tablename=your_pd_dataframe)
>>> com = [{"H": 2, }, {"Po": 1}, {"C": 2}]
>>> a = tmps.convert(com)
```

	0	1	2	3	4	5
0	44.90000	0.00000	0.00000	0.00000	0.00000	0.00000
1	293364.28000	49738.62000	47958.84000	11998.43000	11144.75000	9543.85000

- In default, the proportion would be multiplied in data, also you can neglect weight.

```
>>> tmps = AtomTableMap(search_tp="name", weight=False, tablename=your_pd_dataframe)
>>> com = [{"H": 2, }, {"Po": 1}, {"C": 2}]
>>> a2 = tmps.convert(com)
```

	0	1	2	3	4	5
0	22.45000	0.00000	0.00000	0.00000	0.00000	0.00000
1	293364.28000	49738.62000	47958.84000	11998.43000	11144.75000	9543.85000
2	956.51000	48.16000	19.15000	0.00000	0.00000	0.00000

- Index by structure

```
>>> tmps = AtomTableMap(search_tp="number",tablename="oe.csv")
>>> a = tmps.convert(structure)
```

Note

```
>>> com = [i.species.as_dict() for i in structure.sites]
```

or

```
>>> com =  [{str(i.symbol): 1} for i in structure.species]
```

5.4 Pymatgen Data

The data are using the inner periodical_data.json in pymatgen elemental data.

```
>>> from featurebox.featurizers.atom.mapper import AtomPymatgenPropMap
>>> tmps = AtomPymatgenPropMap(search_tp="name",prop_name = [ "atomic_radius", "atomic_
↪mass", "number", "max_oxidation_state"])
>>> s = [{"H": 2, }, {"Po": 1}, {"C": 2}] # [i.species.as_dict() for i in pymatgen.
↪structure.sites]
>>> a2 = tmps.convert(s) # or
>>> a2 = tmps.convert(structurei)
```

In addition, we could get structure state data by structure.

```
>>> from featurebox.featurizers.state.state_mapper import StructurePymatgenPropMap
>>> tmps = StructurePymatgenPropMap(prop_name = ["density", "volume", "ntypesp"])
>>> a2 = tmps.convert(structurei)
```

This second class is for structure but for atoms, and the first one return the each atom features and the second return the whole feature of structure.

5.5 Polynomial Combination

Polynomial combination.:

```
>>> from featurebox.featurizers.state.union import PolyFeature
>>> n = np.array([[0,1,2,3,4,5],[0.422068,0.360958,0.201433,-0.459164,-0.064783,-0.
↪250939]]).T
>>> ps = pd.DataFrame(n,columns=["f1","f2"],index= ["x0","x1","x2","x3","x4","x5"])
>>> pf = PolyFeature(degree=[1,2])
>>> pf.fit_transform(n)

   f0^1      f1^1  f0^2  f0^1*f1^1      f1^2  ...
0   0.0  0.422068   0.0   0.000000  0.178141  ...
1   1.0  0.360958   1.0   0.360958  0.130291  ...
2   2.0  0.201433   4.0   0.402866  0.040575  ...
3   3.0 -0.459164   9.0  -1.377492  0.210832  ...
4   4.0 -0.064783  16.0  -0.259132  0.004197  ...
5   5.0 -0.250939  25.0  -1.254695  0.062970  ...
```

5.6 Combination

Combination to composition data from element data.:

```
>>> from featurebox.featurizers.atom.mapper import AtomTableMap, AtomJsonMap
>>> data_map = AtomJsonMap(search_tp="name", n_jobs=1)
>>> wa = WeightedAverage(data_map, n_jobs=1,return_type="df")
>>> x3 = [{"H": 2, "Pd": 1},{"He":1,"Al":4}]
>>> wa.fit_transform(x3) # or
```

(continues on next page)

(continued from previous page)

```
>>> wa.fit_transform(structure_list)

         0         1         2   ...        13        14        15
0  0.422068  0.360958  0.201433  ... -0.459164 -0.064783 -0.250939
1  0.007163 -0.471498 -0.072860  ...  0.206306 -0.041006  0.055843

[2 rows x 16 columns]


>>> wa.set_feature_labels(["fea_{}".format(_) for _ in range(16)])
>>> wa.fit_transform(x3)


      fea_0     fea_1     fea_2  ...    fea_13    fea_14    fea_15
0  0.422068  0.360958  0.201433  ... -0.459164 -0.064783 -0.250939
1  0.007163 -0.471498 -0.072860  ...  0.206306 -0.041006  0.055843

[2 rows x 16 columns]
```

5.7 Custom Features

Multiple strategy combinations:

```
>>> ### atom part ####
>>> func_map = [
...    "atomic_radius",
...    "max_oxidation_state",
...    "min_oxidation_state",
...     "atomic_radius_calculated",
...    "critical_temperature",
...    "density_of_solid",
...    "average_ionic_radius",
...    "average_cationic_radius",
...    "average_anionic_radius",]
```

Custom atom Features:

```
>>> from featurebox.featurizers.atom import mapper
>>> from featurebox.featurizers.base_feature import ConverterCat
>>> appa1 = mapper.AtomPymatgenPropMap(prop_name=func_map, search_tp="number")
>>> appa2 = mapper.AtomTableMap(tablename="ele_table.csv", search_tp="number")
>>> appa = ConverterCat(appa1, appa2)
```

Custom state Features:

```
>>> apps1 = state_mapper.StructurePymatgenPropMap(prop_name=["density", "volume",
↪"ntypesp"])
```

Custom bond Features:

```
>>> appb1 = BaseDesGet(nn_strategy="SOAP", numerical_tol=1e-8, cutoff=None, cut_off_
↪name=None)
```

5.8 Use Yourself Data

```
>>> from featurebox.featurizers.atom.mapper import AtomJsonMap
>>> tmps = AtomJsonMap(search_tp="number",embedding_dict="your.json")
>>> tmps = AtomJsonMap(search_tp="number",embedding_dict=Your_dict)
```

```
>>> tmps = AtomTableMap(search_tp="number",tablename="your.csv")
>>> tmps = AtomJsonMap(search_tp="number",tablename=Your_pd_DataFrame)
```

where the search_tp is "number" or "name" depend on your data, but advise use "name" for json data.

5.9 Backforward

Select by Backforward

```
>>> from sklearn.datasets import load_boston
>>> from sklearn.svm import SVR
>>> from featurebox.selection.backforward import BackForward
>>> X,y = load_boston(return_X_y=True)
>>> svr= SVR()
>>> bf = BackForward(svr, primary_feature=4, random_state=1)
>>> new_x = bf.fit_transform(X,y)
>>> bf.support_
>>> array([False, False, False, False, False, False, False, False, False, False,  True,␣
↪False,  True])
```

5.10 Select by Corr

- 1. Corr Automatically

```
>>> from sklearn.datasets import load_boston
>>> from featurebox.selection.corr import Corr
>>> x, y = load_boston(return_X_y=True)
>>> co = Corr(threshold=0.7,multi_index=[0,8],multi_grade=2)
>>> newx = co.fit_transform(x)
>>> print(x.shape)
>>> print(newx.shape)
>>> #(506, 13)
>>> #(506, 9)
```

- 2. Corr Step

```
>>> from sklearn.datasets import load_boston
>>> from featurebox.selection.corr import Corr
>>> x, y = load_boston(return_X_y=True)
>>> co = Corr(threshold=0.7,multi_index=[0,8],multi_grade=2)
```

Nn range [0,8], the features are binding in to 2 sized: [[0,1],[2,3],[4,5],[6,7]] Corresponding to the initial 13 feature. [0,1] -> 0; [2,3] -> 1; [4,5]->2; [6,7]->3; 8->4; 9->5; 10->6; 11->7; 12->8; 13->9;

```
>>> co.fit(x)
>>> Corr(multi_index=[0, 8], threshold=0.7)
>>> group = co.count_cof()
>>> group[1]
>>> #[[0], [1], [2], [3], [4, 5], [4, 5], [6], [7], [8]]
```

In this step, you could select manually, or filter automatically as following.

```
>>> co.remove_coef(group[1]) # Filter automatically by machine.
>>> #[0, 1, 2, 3, 4, 6, 7, 8]
```

where 2 is filtered, Corresponding to the initial feature 14. [0,1] -> 0; [2,3] -> 1; [4,5]->2; [6,7]->3; 8->4; [9->5]; 10->6; 11->7; 12->8; 13->9;

5.11 Name Split

Split complex str compound name to data.

```
>>> from featurebox.data.namesplit import NameSplit
>>> import os
>>> os.chdir(r'.')
>>> name = ['(Ti1.24La3)2',"((Ti1.24)2P2)1H0.2", "((Ti1.24)2)1H0.2", "((Ti1.24))1H0.2",
↪"((Ti)2P2)1H0.2",  "((Ti))1H0.2"]
>>> NSp = NameSplit()
>>> NSp.transform(name)
```

In local disk, there are 2 csv file.

	A	B	C	D	E	F	G	H	I	J
1		Abandon	H	He	Li	Be	B	C	N	O
2	Ti2.48La6	0	0	0	0	0	0	0	0	0
3	Ti2.48P2H0.2	0	0.2	0	0	0	0	0	0	0
4	Ti2.48H0.2	0	0.2	0	0	0	0	0	0	0
5	Ti1.24H0.2	0	0.2	0	0	0	0	0	0	0
6	Ti2P2H0.2	0	0.2	0	0	0	0	0	0	0
7	Ti1H0.2	0	0.2	0	0	0	0	0	0	0

	A	B	C	D	E	F	G	H	I	J	K
1		0	1	2	3	4	5	6	7	8	9
2	Ti2.48La6	Ti	2.48	La	6						
3	Ti2.48P2H0.2	Ti	2.48	P	2	H	0.2				
4	Ti2.48H0.2	Ti	2.48	H	0.2						
5	Ti1.24H0.2	Ti	1.24	H	0.2						
6	Ti2P2H0.2	Ti	2	P	2	H	0.2				
7	Ti1H0.2	Ti	1	H	0.2						

5.12 Batch bader in CMD

This is one sample for get bader in batches.

5.12.1 In CMD

1. Make sure the necessary software and vasp outputs (Just for First Running).

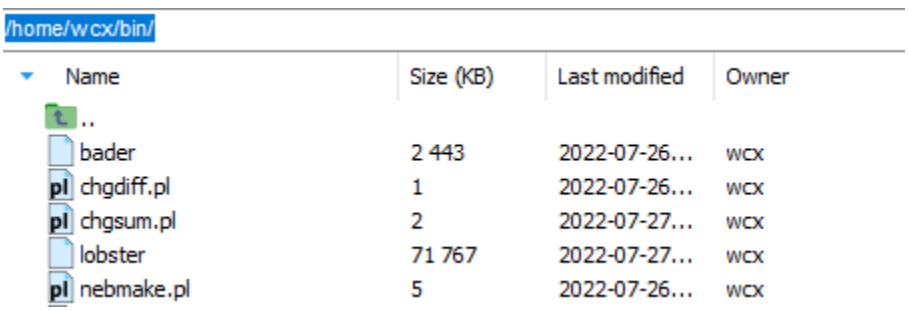
1.1 Set vasp INCAR to calculate:

```
LCHARG = T
LWAVE = T
LAECHG= T
LORBIT= 11
```

1.2 Make sure necessary_files = ["AECCAR0", "AECCAR2", "CHGCAR", "POTCAR", "CONTCAR"] in each case.

1.3 Install sure necessary_software = ["chgsum.pl", "bader"], and with access permission.

Download: Guide: *Command Mode for Extractor*.



2. Get all sub-path by `findpath` command:

```
findpath -if AECCAR2
```

```
(base) [wcx@c0 W2CO2-H]$ findpath -if AECCAR2
Collecting all Paths ...
Filter the Paths ...
Write Out File ...
The 2 paths 'paths.temp' are stored in '/home/wcx/data/W2CO2_add/W2CO2-H'.
OK
(base) [wcx@c0 W2CO2-H]$ fbx bader -j 0
```

3. Run with fbx (featurebox):

```
fbx bader -j 0
```

```
'bader_single.csv' are sored in '/home/wcx/data/W2CO2_add/W2CO2-H/Pt/H/ini_static'
Ok for: /home/wcx/data/W2CO2_add/W2CO2-H/Pt/H/ini_static
100%|
'bader_all.csv' are sored in '/home/wcx/data/W2CO2_add/W2CO2-H'
```

4. Subsequent processing.

```
>>> # Ture to python code.
>>> # More part: The following is not in command model.
>>> # final treatment to extractor need message and formatting.
>>> from featurebox.cli.vasp_bader import BaderStartInter, BaderStartSingleResult
>>> # use  BaderStartInter or BaderStartSingleResult rather than BaderStartZero to␣
↪escape repetition calculation.
>>> bsi = BaderStartInter()
>>> res = pd.read_csv("bader_all.csv")
>>> features = bsi.extract(res, atoms=[0,1,2,3], format_path=None)
>>> print(type(result))  # return one formed pd.Dataframe with necessary message.
```

5.12.2 Traditional python

1. Make sure the necessary software and vasp outputs. (Just for First Running).
2. Get all sub-path (optional).

```
>>> from mgetool.imports.batchfilematch import BatchFileMatch
>>> bfm = BatchFileMatch(".")
>>> bfm.filter_file_name(include="AECCAR2")
>>> paths_list = bfm.merge()
>>> paths_list = bfm.get_leaf_dir(paths_list)
>>> print(paths_list)
```

3. Run with featurebox.

```
>>> from featurebox.cli.vasp_bader import BaderStartZero
>>> bsz = BaderStartZero(n_jobs=4, tq= True, store_single=True)
>>> result = bsz.transform(paths_list)
>>> print(type(result))  # return pd.Dataframe
```

4. Subsequent processing.

```
>>> # More part: The following is not in command model.
>>> # final treatment to extractor need message and formatting.
>>> features = bsz.extract(res, atoms=[0,1,2,3],format_path=None)
>>> print(type(features))  # return one formed pd.Dataframe with necessary message.
```

CHAPTER

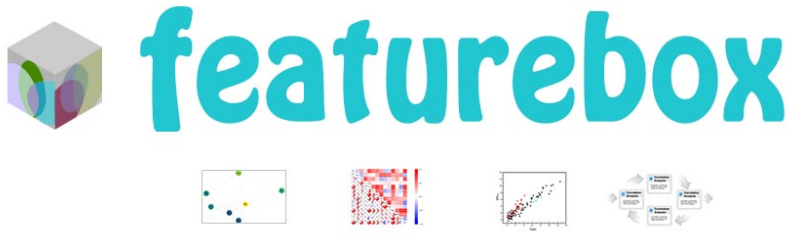
SIX

CONTACT

Thanks for your reading.

This project is one alpha version, if you have question, bugs or writing errors to feedback.

Please contact with 986798607@qq.com.



Featurebox is an open Python library that implements a comprehensive set of machine learning tools for materials informatics, and aims to generate material features quickly and easily.

CHAPTER

SEVEN

FEATURES:

1. Large batch features production.
2. Selection tools with feature binding and restriction.
3. Match with `scikit-learn` and `torch`.

CHAPTER

EIGHT

LINKS

Github | English Version | Chinese Version

CHAPTER

NINE

INDEX

- genindex
- modindex

CHAPTER

TEN

SUPPORT

PYTHON MODULE INDEX

f

INDEX

A

B

E

F

G

H

I

K

M

N

P

Q

R

S

T

U

W