ODDT Documentation

Release 0.1.3

Maciej Wojcikowski

July 27, 2015

Contents

Contents

CHAPTER 1

Installation

1.1 Requirements

- Python 2.7.x
- OpenBabel (2.3.2+) or/and RDKit (2012.03)
- Numpy (1.6.2+)
- Scipy (0.10+)
- Sklearn (0.11+)
- ffnet (0.7.1+) only for neural network functionality.

Note: All installation methods assume that one of toolkits is installed. For detailed installation procedure visit toolkit's website (OpenBabel, RDKit)

Most convenient way of installing ODDT is using PIP. All required python modules will be installed automatically, although toolkits, either OpenBabel (`pip install openbabel`) or RDKit need to be installed manually

```
pip install oddt
```

If you want to install cutting edge version (master branch from GitHub) of ODDT also using PIP

```
pip install git+https://github.com/oddt/oddt.git@master
```

Finally you can install ODDT straight from the source

```
wget https://github.com/oddt/oddt/archive/0.1.1.tar.gz
tar zxvf 0.1.1.tar.gz
cd oddt-0.1.1/
python setup.py install
```

1.2 Common installation problems

ffnet requires numpy.distutils during installation, and you are trying to install ffnet without numpy. You have to install numpy first.

```
pip install numpy
```

Then you can install ODDT

```
pip install oddt
```

CHAPTER 2

Usage Instructions

You can use any supported toolkit united under common API (for reference see Pybel or Cinfony). All methods and software which based on Pybel/Cinfony should be drop in compatible with ODDT toolkits. In contrast to it's predecessors, which were aimed to have minimalistic API, ODDT introduces extended methods and additional handles. This extensions allow to use toolkits at all it's grace and some features may be backported from others to introduce missing functionalities. To name a few:

- coordinates are returned as Numpy Arrays
- atoms and residues methods of Molecule class are lazy, ie. not returning a list of pointers, rather an object which allows indexing and iterating through atoms/residues
- Bond object (similar to Atom)
- *atom_dict*, *ring_dict*, *res_dict* - comprehensive Numpy Arrays containing common information about given entity, particularly useful for high performance computing, ie. interactions, scoring etc.
- lazy Molecule (asynchronous), which is not converted to an object in reading phase, rather passed as a string and read in when underlying object is called
- pickling introduced for Pybel Molecule (internally saved to mol2 string)

2.1 Atom, residues, bonds iteration

One of the most common operation would be iterating through molecules atoms

```
mol = oddt.toolkit.readstring('smi', 'c1cccc1')
for atom in mol:
    print atom.idx
```

Note: mol.atoms, returns an object (`AtomStack`) which can be access via indexes or iterated

Iterating over residues is also very convenient, especially for proteins

```
for res in mol.residues:
    print res.name
```

Additionally residues can fetch atoms belonging to them:

```
for res in mol.residues:
    for atom in res:
        print atom.idx
```

Bonds are also iterable, similar to residues:

```
for bond in mol.bonds:
    print bond.order
    for atom in bond:
        print atom.idx
```

2.2 Reading molecules

Reading molecules is mostly identical to Pybel.

Reading from file

```
for mol in oddt.toolkit.readfile('smi', 'test.smi'):
    print mol.title
```

Reading from string

```
mol = oddt.toolkit.readstring('smi', 'c1ccccc1 benzene'):
    print mol.title
```

Note: You can force molecules to be read in asynchronously, aka "lazy molecules". Current default is not to produce lazy molecules due to OpenBabel's Memory Leaks in OBConverter. Main advantage of lazy molecules is using them in multiprocessing, then conversion is spreaded on all jobs.

Reading molecules from file in asynchronous manner

```
for mol in oddt.toolkit.readfile('smi', 'test.smi', lazy=True):
    pass
```

This example will execute instantaneously, since no molecules were evaluated.

2.3 Numpy Dictionaries - store your molecule as an uniform structure

Most important and handy property of Molecule in ODDT are Numpy dictionaries containing most properties of supplied molecule. Some of them are straightforward, other require some calculation, ie. atom features. Dictionaries are provided for major entities of molecule: atoms, bonds, residues and rings. It was primarily used for interactions calculations, although it is applicable for any other calculation. The main benefit is marvelous Numpy broadcasting and subsetting.

Each dictionary is defined as a format in Numpy.

2.3.1 atom_dict

Atom basic information

- '*coords*', type: `float16`, shape: (3) - atom coordinates
- '*charge*', type: `float16` - atom's charge
- '*atomicnum*', type: `int8` - atomic number
- '*atomtype', type: `a4` - Sybyl atom's type
- '*hybridization*', type: `int8` - atoms hybrydization

- '*neighbors*', type: `float16`, shape: (4,3) - coordinates of non-H neighbors coordinates for angles (max of 4 neighbors should be enough)

Residue information for current atom

- '*resid*', type: `int16` - residue ID
- '*resname*', type: `a3` - Residue name (3 letters)
- '*isbackbone*', type: `bool` - is atom part of backbone

Atom properties

- '*isacceptor*', type: `bool` - is atom H-bond acceptor
- '*isdonor*', type: `bool` - is atom H-bond donor
- '*isdonorh*', type: `bool` - is atom H-bond donor Hydrogen
- '*ismetal*', type: `bool` - is atom a metal
- '*ishydrophobe*', type: `bool` - is atom hydrophobic
- '*isaromatic*', type: `bool` - is atom aromatic
- '*isminus*', type: `bool` - is atom negatively charged/chargable
- '*isplus*', type: `bool` - is atom positively charged/chargable
- '*ishalogen*', type: `bool` - is atom a halogen

Secondary structure

- '*isalpha*', type: `bool` - is atom a part of alpha helix
- '*isbeta*', type: `bool'` - is atom a part of beta strand

2.3.2 ring_dict

- '*centroid*', type: `float16`, shape: 3 - coordinates of ring's centroid
- '*vector*', type: `float16`, shape: 3 - normal vector for ring
- '*isalpha*', type: `bool` - is ring a part of alpha helix
- '*isbeta*', type: `bool'` - is ring a part of beta strand

2.3.3 res_dict

- '*id*', type: `int16` - residue ID
- '*resname*', type: `a3` - Residue name (3 letters)
- '*N*', type: `float16`, shape: 3 - cordinates of backbone N atom
- '*CA*', type: `float16`, shape: 3 - cordinates of backbone CA atom
- '*C*', type: `float16`, shape: 3 - cordinates of backbone C atom
- '*isalpha*', type: `bool` - is residue a part of alpha helix
- '*isbeta*', type: `bool'` - is residue a part of beta strand

Note: All aforementioned dictionaries are generated "on demand", and are cached for molecule, thus can be shared between calculations. Caching of dictionaries brings incredible performance gain, since in some applications their generation is the major time consuming task.

Get all acceptor atoms:

```
mol.atom_dict['is_acceptor']
```

CHAPTER 3

ODDT API documentation

3.1 oddt package

3.1.1 Subpackages

oddt.docking package

Submodules

oddt.docking.autodock_vina module

class oddt.docking.autodock_vina.**autodock_vina**(*protein=None*, *auto_ligand=None*, *size=(10, 10, 10)*, *center=(0, 0, 0)*, *exhaustivness=8*, *num_modes=9*, *energy_range=3*, *seed=None*, *prefix_dir='/tmp'*, *n_cpu=1*, *executable=None*, *autocleanup=True*)

Bases: object

Autodock Vina docking engine, which extends it's capabilities: automatic box (autocentering on ligand).

Parameters **protein: oddt.toolkit.Molecule object (default=None)**

Protein object to be used while generating descriptors.

auto_ligand: oddt.toolkit.Molecule object or string (default=None) Ligand use to center the docking box. Either ODDT molecule or a file (opened based on extesion and read to ODDT molecule). Box is centered on geometric center of molecule.

size: tuple, shape=[3] (default=(10,10,10)) Dimentions of docking box (in Angstroms)

center: tuple, shape=[3] (default=(0,0,0)) The center of docking box in cartesian space.

exhaustiveness: int (default=8) Exhaustiveness parameter of Autodock Vina

num_modes: int (default=9) Number of conformations generated by Autodock Vina

energy_range: int (default=3) Energy range cutoff for Autodock Vina

seed: int or None (default=None) Random seed for Autodock Vina

prefix_dir: string (default=/tmp) Temporary directory for Autodock Vina files

executable: string or None (default=None) Autodock Vina executable location in the system. It's realy necessary if autodetection fails.

autocleanup: bool (default=True) Should the docking engine clean up after execution?

Attributes

tmp_dir

Methods

clean()	
dock(ligands[, protein, single])	Automated docking procedure.
score(ligands[, protein, single])	Automated scoring procedure.
set_protein(protein)	Change protein to dock to.

clean()

dock(*ligands*, *protein=None*, *single=False*)
Automated docking procedure.

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to dock

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

single: bool (default=False) A flag to indicate single ligand docking (performance reasons (eg. there is no need for subdirectory for one ligand)

Returns **ligands** : array of oddt.toolkit.Molecule objects

Array of ligands (scores are stored in mol.data method)

score(*ligands*, *protein=None*, *single=False*)
Automated scoring procedure.

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to score

protein: oddt.toolkit.Molecule object or None Protein object to be used. If None, then the default one is used, else the protein is new default.

single: bool (default=False) A flag to indicate single ligand scoring (performance reasons (eg. there is no need for subdirectory for one ligand)

Returns **ligands** : array of oddt.toolkit.Molecule objects

Array of ligands (scores are stored in mol.data method)

set_protein(*protein*)
Change protein to dock to.

Parameters **protein: oddt.toolkit.Molecule object**

Protein object to be used.

tmp_dir

oddt.docking.autodock_vina.**parse_vina_docking_output**(*output*)
Function parsing Autodock Vina docking output to a dictionary

Parameters **output** : string

Autodock Vina standard ouptud (STDOUT).

Returns **out** : dict

dicitionary containing scores computed by Autodock Vina

oddt.docking.autodock_vina.**parse_vina_scoring_output**(*output*)
Function parsing Autodock Vina scoring output to a dictionary

Parameters **output** : string

Autodock Vina standard ouptud (STDOUT).

Returns **out** : dict

dicitionary containing scores computed by Autodock Vina

oddt.docking.autodock_vina.**random**() → x in the interval [0, 1).

Module contents

oddt.scoring package

Subpackages

oddt.scoring.descriptors package

Submodules

oddt.scoring.descriptors.binana module Internal implementation of binana software (http://nbcr.ucsd.edu/data/sw/hosted/binana/)

class oddt.scoring.descriptors.binana.**binana_descriptor**(*protein=None*)
Bases: object

Descriptor build from binana script (as used in NNScore 2.0

Parameters **protein: oddt.toolkit.Molecule object (default=None)**

Protein object to be used while generating descriptors.

Continued on next page

Table 3.3 – continued from previous page

Methods

build(ligands[, protein])	Descriptor building method
set_protein(protein)	One function to change all relevant proteins

build (*ligands*, *protein=None*)
Descriptor building method

Parameters **ligands: array-like**

An array of generator of oddt.toolkit.Molecule objects for which the descriptor is computed

protein: oddt.toolkit.Molecule object (default=None) Protein object to be used while generating descriptors. If none, then the default protein (from constructor) is used. Otherwise, protein becomes new global and default protein.

Returns descs: numpy array, shape=[n_samples, 351]

An array of binana descriptors, aligned with input ligands

set_protein (*protein*)
One function to change all relevant proteins

Parameters **protein: oddt.toolkit.Molecule object**

Protein object to be used while generating descriptors. Protein becomes new global and default protein.

Module contents

oddt.scoring.descriptors.**atoms_by_type** (*atom_dict*, *types*, *mode='atomic_nums'*)

Returns atom dictionaries based on given criteria. Currently we have 3 types of atom selection criteria:

- atomic numbers ['atomic_nums']
- Sybyl Atom Types ['atom_types_sybyl']
- AutoDock4 atom types ['atom_types_ad4'] (http://autodock.scripps.edu/faqs-help/faq/where-do-i-set-the-autodock-4-force-field-parameters)

Parameters **atom_dict: oddt.toolkit.Molecule.atom_dict**

Atom dictionary as implemeted in oddt.toolkit.Molecule class

types: array-like List of atom types/numbers wanted.

Returns out: dictionary of shape=[len(types)]

A dictionary of queried atom types (types are keys of the dictionary). Values are of oddt.toolkit.Molecule.atom_dict type.

class oddt.scoring.descriptors.**close_contacts** (*protein=None*, *cutoff=4*, *mode='atomic_nums'*, *ligand_types=None*, *protein_types=None*, *aligned_pairs=False*)

Bases: object

Close contacts descriptor which tallies atoms of type X in certain cutoff from atoms of type Y.

Parameters **protein: oddt.toolkit.Molecule or None (default=None)**

Default protein to use as reference

cutoff: int (default=4) Cutoff for atoms in Angstroms

mode: string (default='atomic_nums') Method of atoms selection, as used in *atoms_by_type*

ligand_types: array List of ligand atom types to use

protein_types: array List of protein atom types to use

aligned_pairs: bool (default=False) Flag indicating should permutation of types should be done, otherwise the atoms are treated as aligned pairs.

Methods

build(ligands[, protein, single])	Builds descriptors for series of ligands

build (*ligands*, *protein=None*, *single=False*)

Builds descriptors for series of ligands

Parameters **ligands: iterable of oddt.toolkit.Molecules or oddt.toolkit.Molecule**

A list or iterable of ligands to build the descriptor or a single molecule.

protein: oddt.toolkit.Molecule or None (default=None) Default protein to use as reference

single: bool (default=False) Flag indicating if the ligand is single.

class oddt.scoring.descriptors.**fingerprints** (*fp='fp2'*, *toolkit='ob'*)

Bases: object

Methods

build(mols[, single])	

build (*mols*, *single=False*)

oddt.scoring.functions package

Submodules

oddt.scoring.functions.NNScore module

oddt.scoring.functions.RFScore module

Module contents

oddt.scoring.models package

Submodules

oddt.scoring.models.classifiers module

oddt.scoring.models.classifiers.**randomforest**
alias of RandomForestClassifier

oddt.scoring.models.classifiers.**svm**
alias of SVC

oddt.scoring.models.neuralnetwork module

oddt.scoring.models.regressors module

Module contents

Module contents

oddt.scoring.**cross_validate**(*model*, *cv_set*, *cv_target*, *n=10*, *shuffle=True*, *n_jobs=1*)
Perform cross validation of model using provided data

Parameters **model: object**

Model to be tested

cv_set: array-like of shape = [n_samples, n_features] Estimated target values.

cv_target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

n: integer (default = 10) How many folds to be created from dataset

shuffle: bool (default = True) Should data be shuffled before folding.

n_jobs: integer (default = 1) How many CPUs to use during cross validation

Returns r2: array of shape = [n]

R^2 score for each of generated folds

class oddt.scoring.**ensemble_model**(*models*)
Bases: object

Proxy class to build an ensemble of models with an API as one

Parameters **models: array**

An array of models

Methods

fit(X, y, *args, **kwargs)	
predict(X, *args, **kwargs)	
score(X, y, *args, **kwargs)	

fit (*X*, *y*, **args*, ***kwargs*)

predict (*X*, **args*, ***kwargs*)

score (*X*, *y*, **args*, ***kwargs*)

class oddt.scoring.**scorer** (*model_instances*, *descriptor_generator_instances*, *score_title='score'*)
Bases: object

Scorer class is parent class for scoring functions. It's capable of using multiple models and/or multiple descriptors. If multiple models and multiple descriptors are used they should be aligned, since no permutation of such is made.

Parameters **model_instances: array of models**

An array of medels compatible with sklearn API (fit, predict and score methods)

descriptor_generator_instances: array of descriptors An array of descriptor objects

score_title: string Title of score to be used.

Methods

fit(ligands, target, *args, **kwargs)	Trains model on supplied ligands and target values
load(filename)	Loads scoring function from a pickle file.
predict(ligands, *args, **kwargs)	Predicts values (eg.
predict_ligand(ligand)	Local method to score one ligand and update it's scores.
predict_ligands(ligands)	Method to score ligands lazily
save(filename)	Saves scoring function to a pickle file.
score(ligands, target, *args, **kwargs)	Methods estimates the quality of prediction as squared correlation coefficient (R^2)
set_protein(protein)	Proxy method to update protein in all relevant places.

fit (*ligands*, *target*, **args*, ***kwargs*)
Trains model on supplied ligands and target values

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

classmethod load (*filename*)
Loads scoring function from a pickle file.

Parameters **filename: string**

Pickle filename

Returns sf: scorer-like object

Scoring function object loaded from a pickle

predict (*ligands*, **args*, ***kwargs*)
Predicts values (eg. affinity) for supplied ligands

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns predicted: np.array or array of np.arrays of shape = [n_ligands]

Predicted scores for ligands

predict_ligand (*ligand*)
Local method to score one ligand and update it's scores.

Parameters **ligand: oddt.toolkit.Molecule object**

Ligand to be scored

Returns ligand: oddt.toolkit.Molecule object

Scored ligand with updated scores

predict_ligands (*ligands*)
Method to score ligands lazily

Parameters **ligands: iterable of oddt.toolkit.Molecule objects**

Ligands to be scored

Returns ligand: iterator of oddt.toolkit.Molecule objects

Scored ligands with updated scores

save (*filename*)
Saves scoring function to a pickle file.

Parameters **filename: string**

Pickle filename

score (*ligands*, *target*, **args*, ***kwargs*)
Methods estimates the quality of prediction as squared correlation coefficient (R^2)

Parameters **ligands: array-like of ligands**

Ground truth (correct) target values.

target: array-like of shape = [n_samples] or [n_samples, n_outputs] Estimated target values.

Returns r2: float

Squared correlation coefficient (R^2) for prediction

set_protein (*protein*)
Proxy method to update protein in all relevant places.

Parameters **protein: oddt.toolkit.Molecule object**

New default protein

oddt.toolkits package

Submodules

oddt.toolkits.ob module

class oddt.toolkits.ob.**Residue**(*OBResidue*)
Bases: object

Represent a Pybel residue.

Required parameter: OBResidue – an Open Babel OBResidue

Attributes: atoms, idx, name.

(refer to the Open Babel library documentation for more info).

The original Open Babel atom can be accessed using the attribute: OBResidue

Attributes

atoms
idx
name

atoms

idx

name

oddt.toolkits.ob.**pickle_mol**(*self*)

oddt.toolkits.ob.**readfile**(*format*, *filename*, *opt=None*, *lazy=False*)

oddt.toolkits.ob.**unpickle_mol**(*source*)

oddt.toolkits.rdk module

rdkit - A Cinfony module for accessing the RDKit from CPython

Global variables: Chem and AllChem - the underlying RDKit Python bindings informats - a dictionary of supported input formats outformats - a dictionary of supported output formats descs - a list of supported descriptors fps - a list of supported fingerprint types forcefields - a list of supported forcefields

class oddt.toolkits.rdk.**Atom**(*Atom*)
Bases: object

Represent an rdkit Atom.

Required parameters: Atom – an RDKit Atom

Attributes: atomicnum, coords, formalcharge

The original RDKit Atom can be accessed using the attribute: Atom

Attributes

atomicnum
coords
formalcharge
idx
neighbors
partialcharge

atomicnum

coords

formalcharge

idx

neighbors

partialcharge

class oddt.toolkits.rdk.**Fingerprint**(*fingerprint*)
Bases: object

A Molecular Fingerprint.

Required parameters: fingerprint – a vector calculated by one of the fingerprint methods

Attributes: fp – the underlying fingerprint object bits – a list of bits set in the Fingerprint

Methods: The "|" operator can be used to calculate the Tanimoto coeff. For example, given two Fingerprints 'a', and 'b', the Tanimoto coefficient is given by:

tanimoto = a | b

Attributes

raw

raw

class oddt.toolkits.rdk.**Molecule**(*Mol=None*, *source=None*, *protein=False*)
Bases: object

Represent an rdkit Molecule.

Required parameter: Mol – an RDKit Mol or any type of cinfony Molecule

Attributes: atoms, data, formula, molwt, title

Methods: addh(), calcfp(), calcdesc(), draw(), localopt(), make3D(), removeh(), write()

The underlying RDKit Mol can be accessed using the attribute: Mol

Attributes

Mol
atom_dict

Continued on next page

Table 3.11 – continued from previous page

atoms	
canonic_order	Returns np.array with canonic order of heavy atoms in the molecule
charges	
clone	
coords	
data	
formula	
molwt	
num_rotors	
res_dict	
ring_dict	
sssr	
title	

Methods

addh()	Add hydrogens.
calcdesc([descnames])	Calculate descriptor values.
calcfp([fptype, opt])	Calculate a molecular fingerprint.
clone_coords(source)	
draw([show, filename, update, usecoords])	Create a 2D depiction of the molecule.
localopt([forcefield, steps])	Locally optimize the coordinates.
make3D([forcefield, steps])	Generate 3D coordinates.
removeh()	Remove hydrogens.
write([format, filename, overwrite])	Write the molecule to a file or return a string.

Mol

addh()
Add hydrogens.

atom_dict

atoms

calcdesc(*descnames=[]*)
Calculate descriptor values.

Optional parameter: descnames – a list of names of descriptors

If descnames is not specified, all available descriptors are calculated. See the descs variable for a list of available descriptors.

calcfp(*fptype='rdkit'*, *opt=None*)
Calculate a molecular fingerprint.

Optional parameters:

fptype – the fingerprint type (default is "rdkit"). See the fps variable for a list of of available fingerprint types.

opt – a dictionary of options for fingerprints. Currently only used for radius and bitInfo in Morgan fingerprints.

canonic_order
Returns np.array with canonic order of heavy atoms in the molecule

charges

clone

clone_coords(*source*)

coords

data

draw(*show=True*, *filename=None*, *update=False*, *usecoords=False*)
Create a 2D depiction of the molecule.

Optional parameters: show – display on screen (default is True) filename – write to file (default is None) update – update the coordinates of the atoms to those

determined by the structure diagram generator (default is False)

usecoords – don't calculate 2D coordinates, just use the current coordinates (default is False)

Aggdraw or Cairo is used for 2D depiction. Tkinter and Python Imaging Library are required for image display.

formula

localopt(*forcefield='uff'*, *steps=500*)
Locally optimize the coordinates.

Optional parameters:

forcefield – default is "uff". See the forcefields variable for a list of available forcefields.

steps – default is 500

If the molecule does not have any coordinates, make3D() is called before the optimization.

make3D(*forcefield='uff'*, *steps=50*)
Generate 3D coordinates.

Optional parameters:

forcefield – default is "uff". See the forcefields variable for a list of available forcefields.

steps – default is 50

Once coordinates are generated, a quick local optimization is carried out with 50 steps and the UFF forcefield. Call localopt() if you want to improve the coordinates further.

molwt

num_rotors

removeh()
Remove hydrogens.

res_dict

ring_dict

sssr

title

write(*format='smi'*, *filename=None*, *overwrite=False*, ***kwargs*)
Write the molecule to a file or return a string.

Optional parameters:

format – see the informats variable for a list of available output formats (default is "smi")

filename – default is None overwite – if the output file already exists, should it

be overwritten? (default is False)

If a filename is specified, the result is written to a file. Otherwise, a string is returned containing the result.

To write multiple molecules to the same file you should use the Outputfile class.

class oddt.toolkits.rdk.**MoleculeData**(*Mol*)
Bases: object

Store molecule data in a dictionary-type object

Required parameters: Mol – an RDKit Mol

Methods and accessor methods are like those of a dictionary except that the data is retrieved on-the-fly from the underlying Mol.

Example: >>> mol = readfile("sdf", 'head.sdf').next() >>> data = mol.data >>> print data {'Comment': 'CORINA 2.61 0041 25.10.2001', 'NSC': '1'} >>> print len(data), data.keys(), data.has_key("NSC") 2 ['Comment', 'NSC'] True >>> print data['Comment'] CORINA 2.61 0041 25.10.2001 >>> data['Comment'] = 'This is a new comment' >>> for k,v in data.iteritems(): ... print k, "–>", v Comment –> This is a new comment NSC –> 1 >>> del data['NSC'] >>> print len(data), data.keys(), data.has_key("NSC") 1 ['Comment'] False

Methods

clear()
has_key(key)
items()
iteritems()
keys()
update(dictionary)
values()

clear()

has_key(*key*)

items()

iteritems()

keys()

update(*dictionary*)

values()

class oddt.toolkits.rdk.**Outputfile**(*format*, *filename*, *overwrite=False*)
Bases: object

Represent a file to which *output* is to be sent.

Required parameters:

format - see the outformats variable for a list of available output formats

filename

Optional parameters:

overwite – if the output file already exists, should it be overwritten? (default is False)

Methods: write(molecule) close()

Methods

close()	Close the Outputfile to further writing.
write(molecule)	Write a molecule to the output file.

close()
Close the Outputfile to further writing.

write(*molecule*)
Write a molecule to the output file.

Required parameters: molecule

class oddt.toolkits.rdk.**Smarts**(*smartspattern*)
Bases: object

Initialise with a SMARTS pattern.

Methods

findall(molecule)	Find all matches of the SMARTS pattern to a particular molecule.

findall(*molecule*)
Find all matches of the SMARTS pattern to a particular molecule.

Required parameters: molecule

oddt.toolkits.rdk.**base_feature_factory = <MagicMock name='mock.Chem.AllChem.BuildFeatureFactory()' id='**
Global feature factory based on BaseFeatures.fdef

oddt.toolkits.rdk.**descs = []**
A list of supported descriptors

oddt.toolkits.rdk.**forcefields = ['uff']**
A list of supported forcefields

oddt.toolkits.rdk.**fps = ['rdkit', 'layered', 'maccs', 'atompairs', 'torsions', 'morgan']**
A list of supported fingerprint types

oddt.toolkits.rdk.**informats = {'inchi': 'InChI', 'mol2': 'Tripos MOL2 file', 'sdf': 'MDL SDF file', 'smi': 'SMILES',**
A dictionary of supported input formats

oddt.toolkits.rdk.**outformats = {'inchikey': 'InChIKey', 'sdf': 'MDL SDF file', 'can': 'Canonical SMILES', 'smi': 'S**
A dictionary of supported output formats

oddt.toolkits.rdk.**readfile**(*format*, *filename*, **args*, ***kwargs*)
Iterate over the molecules in a file.

Required parameters:

format - see the informats variable for a list of available input formats

filename

You can access the first molecule in a file using the next() method of the iterator:

mol = readfile("smi", "myfile.smi").next()

You can make a list of the molecules in a file using: mols = list(readfile("smi", "myfile.smi"))

You can iterate over the molecules in a file as shown in the following code snippet: >>> atomtotal = 0 >>> for mol in readfile("sdf", "head.sdf"): ... atomtotal += len(mol.atoms) ... >>> print atomtotal 43

`oddt.toolkits.rdk.`**`readstring`**(*format*, *string*, ***kwargs*)
Read in a molecule from a string.

Required parameters:

format - see the informats variable for a list of available input formats

string

Example: >>> input = "C1=CC=CS1" >>> mymol = readstring("smi", input) >>> len(mymol.atoms) 5

Module contents

3.1.2 Submodules

3.1.3 oddt.interactions module

Module calculates interactions between two molecules (proein-protein, protein-ligand, small-small). Currently following interacions are implemented:

- hydrogen bonds
- halogen bonds
- pi stacking (parallel and perpendicular)
- salt bridges
- hydrophobic contacts
- pi-cation
- metal coordination
- pi-metal

`oddt.interactions.`**`close_contacts`**(*x*, *y*, *cutoff*, *x_column='coords'*, *y_column='coords'*)
Returns pairs of atoms which are within close contac distance cutoff.

Parameters **x, y** : atom_dict-type numpy array

Atom dictionaries generated by oddt.toolkit.Molecule objects.

cutoff [float] Cutoff distance for close contacts

x_column, ycolumn [string, (default='coords')] Column containing coordinates of atoms (or pseudo-atoms, i.e. ring centroids)

Returns **x_, y_** : atom_dict-type numpy array

Aligned pairs of atoms in close contact for further processing.

oddt.interactions.**hbond_acceptor_donor**(*mol1*, *mol2*, *cutoff=3.5*, *base_angle=120*, *tolerance=30*)

Returns pairs of acceptor-donor atoms, which meet H-bond criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute H-bond acceptor and H-bond donor pairs

cutoff [float, (default=3.5)] Distance cutoff for A-D pairs

base_angle [int, (default=120)] Base angle determining allowed direction of hydrogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which H-bonds are considered as strict.

Returns **a, d** : atom_dict-type numpy array

Aligned arrays of atoms forming H-bond, firstly acceptors, secondly donors.

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' H-bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**hbond**(*mol1*, *mol2*, **args*, ***kwargs*)

Calculates H-bonds between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute H-bond acceptor and H-bond donor pairs

cutoff [float, (default=3.5)] Distance cutoff for A-D pairs

base_angle [int, (default=120)] Base angle determining allowed direction of hydrogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which H-bonds are considered as strict.

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming H-bond

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' H-bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**halogenbond_acceptor_halogen**(*mol1*, *mol2*, *base_angle_acceptor=120*, *base_angle_halogen=180*, *tolerance=30*, *cutoff=4*)

Returns pairs of acceptor-halogen atoms, which meet halogen bond criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute halogen bond acceptor and halogen pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

base_angle_acceptor [int, (default=120)] Base angle determining allowed direction of halogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

base_angle_halogen [int (default=180)] Ideal base angle between halogen bond and halogen-neighbor bond

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which halogen bonds are considered as strict.

Returns **a, h** : atom_dict-type numpy array

Aligned arrays of atoms forming halogen bond, firstly acceptors, secondly halogens

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' halogen bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**halogenbond**(*mol1*, *mol2*, ***kwargs*)

Calculates halogen bonds between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute halogen bond acceptor and halogen pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

base_angle_acceptor [int, (default=120)] Base angle determining allowed direction of halogen bond formation, which is devided by the number of neighbors of acceptor atom to establish final directional angle

base_angle_halogen [int (default=180)] Ideal base angle between halogen bond and halogen-neighbor bond

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in which halogen bonds are considered as strict.

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming halogen bond

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' halogen bond (pass all angular cutoffs). If false, only distance cutoff is met, therefore the bond is 'crude'.

oddt.interactions.**pi_stacking**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of rings, which meet pi stacking criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute ring pairs

cutoff [float, (default=5)] Distance cutoff for Pi-stacking pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (parallel or perpendicular) in which pi-stackings are considered as strict.

Returns **r1, r2** : ring_dict-type numpy array

Aligned arrays of rings forming pi-stacking

strict_parallel [numpy array, dtype=bool] Boolean array align with ring pairs, informing whether rings form 'strict' parallel pi-stacking. If false, only distance cutoff is met, therefore the stacking is 'crude'.

strict_perpendicular [numpy array, dtype=bool] Boolean array align with ring pairs, informing whether rings form 'strict' perpendicular pi-stacking (T-shaped, T-face, etc.). If false, only distance cutoff is met, therefore the stacking is 'crude'.

oddt.interactions.**salt_bridge_plus_minus**(*mol1*, *mol2*, *cutoff=4*)
Returns pairs of plus-mins atoms, which meet salt bridge criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute plus and minus pairs

cutoff [float, (default=4)] Distance cutoff for A-H pairs

Returns **plus, minus** : atom_dict-type numpy array

Aligned arrays of atoms forming salt bridge, firstly plus, secondly minus

oddt.interactions.**salt_bridges**(*mol1*, *mol2*, **args*, ***kwargs*)
Calculates salt bridges between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute plus and minus pairs

cutoff [float, (default=4)] Distance cutoff for plus-minus pairs

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming salt bridges

oddt.interactions.**hydrophobic_contacts**(*mol1*, *mol2*, *cutoff=4*)
Calculates hydrophobic contacts between molecules

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute hydrophobe pairs

cutoff [float, (default=4)] Distance cutoff for hydrophobe pairs

Returns **mol1_atoms, mol2_atoms** : atom_dict-type numpy array

Aligned arrays of atoms forming hydrophobic contacts

oddt.interactions.**pi_cation**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)
Returns pairs of ring-cation atoms, which meet pi-cation criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute ring-cation pairs

cutoff [float, (default=5)] Distance cutoff for Pi-cation pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (perpendicular) in which pi-cation are considered as strict.

Returns **r1** : ring_dict-type numpy array

Aligned rings forming pi-stacking

plus2 [atom_dict-type numpy array] Aligned cations forming pi-cation

strict_parallel [numpy array, dtype=bool] Boolean array align with ring-cation pairs, informing whether they form 'strict' pi-cation. If false, only distance cutoff is met, therefore the interaction is 'crude'.

oddt.interactions.**acceptor_metal**(*mol1*, *mol2*, *base_angle=120*, *tolerance=30*, *cutoff=4*)

Returns pairs of acceptor-metal atoms, which meet metal coordination criteria Note: This function is directional (mol1 holds acceptors, mol2 holds metals)

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute acceptor and metal pairs

cutoff [float, (default=4)] Distance cutoff for A-M pairs

base_angle [int, (default=120)] Base angle determining allowed direction of metal coordination, which is devided by the number of neighbors of acceptor atom to establish final directional angle

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (base_angle/n_neighbors) in metal coordination are considered as strict.

Returns **a, d** : atom_dict-type numpy array

Aligned arrays of atoms forming metal coordination, firstly acceptors, secondly metals.

strict [numpy array, dtype=bool] Boolean array align with atom pairs, informing whether atoms form 'strict' metal coordination (pass all angular cutoffs). If false, only distance cutoff is met, therefore the interaction is 'crude'.

oddt.interactions.**pi_metal**(*mol1*, *mol2*, *cutoff=5*, *tolerance=30*)

Returns pairs of ring-metal atoms, which meet pi-metal criteria

Parameters **mol1, mol2** : oddt.toolkit.Molecule object

Molecules to compute ring-metal pairs

cutoff [float, (default=5)] Distance cutoff for Pi-metal pairs

tolerance [int, (default=30)] Range (+/- tolerance) from perfect direction (perpendicular) in which pi-metal are considered as strict.

Returns **r1** : ring_dict-type numpy array

Aligned rings forming pi-metal

m [atom_dict-type numpy array] Aligned metals forming pi-metal

strict_parallel [numpy array, dtype=bool] Boolean array align with ring-metal pairs, informing whether they form 'strict' pi-metal. If false, only distance cutoff is met, therefore the interaction is 'crude'.

3.1.4 oddt.metrics module

Metrics for estimating performance of drug discovery methods implemented in ODDT

oddt.metrics.**roc**(*y_true*, *y_score*, *pos_label=None*, *sample_weight=None*)

Compute Receiver operating characteristic (ROC)

Note: this implementation is restricted to the binary classification task.

Parameters **y_true** : array, shape = [n_samples]

True binary labels in range {0, 1} or {-1, 1}. If labels are not binary, pos_label should be explicitly given.

y_score : array, shape = [n_samples]

Target scores, can either be probability estimates of the positive class or confidence values.

pos_label : int

Label considered as positive and others are considered negative.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returns **fpr** : array, shape = [>2]

Increasing false positive rates such that element i is the false positive rate of predictions with score >= thresholds[i].

tpr : array, shape = [>2]

Increasing true positive rates such that element i is the true positive rate of predictions with score >= thresholds[i].

thresholds : array, shape = [n_thresholds]

Decreasing thresholds on the decision function used to compute fpr and tpr. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(y_score) + 1*.

See also:

roc_auc_score Compute Area Under the Curve (AUC) from prediction scores

Notes

Since the thresholds are sorted from low to high values, they are reversed upon returning them to ensure they correspond to both `fpr` and `tpr`, which are sorted in reversed order during their calculation.

References

[R1]

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, scores, pos_label=2)
>>> fpr
array([ 0. ,  0.5,  0.5,  1. ])
>>> tpr
array([ 0.5,  0.5,  1. ,  1. ])
>>> thresholds
array([ 0.8 ,  0.4 ,  0.35,  0.1 ])
```

oddt.metrics.**auc**(*x*, *y*, *reorder=False*)

Compute Area Under the Curve (AUC) using the trapezoidal rule

This is a general function, given points on a curve. For computing the area under the ROC-curve, see `roc_auc_score()`.

Parameters **x** : array, shape = [n]

x coordinates.

y : array, shape = [n]

y coordinates.

reorder : boolean, optional (default=False)

If True, assume that the curve is ascending in the case of ties, as for an ROC curve. If the curve is non-ascending, the result will be wrong.

Returns **auc** : float

See also:

roc_auc_score Computes the area under the ROC curve

precision_recall_curve Compute precision-recall pairs for different probability thresholds

Examples

```
>>> import numpy as np
>>> from sklearn import metrics
>>> y = np.array([1, 1, 2, 2])
>>> pred = np.array([0.1, 0.4, 0.35, 0.8])
>>> fpr, tpr, thresholds = metrics.roc_curve(y, pred, pos_label=2)
>>> metrics.auc(fpr, tpr)
0.75
```

oddt.metrics.**roc_auc**(*y_true*, *y_score*, *average='macro'*, *sample_weight=None*)

Compute Area Under the Curve (AUC) from prediction scores

Note: this implementation is restricted to the binary classification task or multilabel classification task in label indicator format.

Parameters **y_true** : array, shape = [n_samples] or [n_samples, n_classes]

True binary labels in binary label indicators.

y_score : array, shape = [n_samples] or [n_samples, n_classes]

Target scores, can either be probability estimates of the positive class, confidence values, or binary decisions.

average : string, [None, 'micro', 'macro' (default), 'samples', 'weighted']

If `None`, the scores for each class are returned. Otherwise, this determines the type of averaging performed on the data:

'micro': Calculate metrics globally by considering each element of the label indicator matrix as a label.

'macro': Calculate metrics for each label, and find their unweighted mean. This does not take label imbalance into account.

'weighted': Calculate metrics for each label, and find their average, weighted by support (the number of true instances for each label).

'samples': Calculate metrics for each instance, and find their average.

sample_weight : array-like of shape = [n_samples], optional

Sample weights.

Returns **auc** : float

See also:

average_precision_score Area under the precision-recall curve

roc_curve Compute Receiver operating characteristic (ROC)

References

[R2]

Examples

```
>>> import numpy as np
>>> from sklearn.metrics import roc_auc_score
>>> y_true = np.array([0, 0, 1, 1])
>>> y_scores = np.array([0.1, 0.4, 0.35, 0.8])
>>> roc_auc_score(y_true, y_scores)
0.75
```

oddt.metrics.**roc_log_auc**(*y_true*, *y_score*, *pos_label=None*, *log_min=0.001*, *log_max=1.0*)

Computes area under semi-log ROC for random distribution.

Parameters **y_true** : array, shape=[n_samples]

True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

pos_label: int Positive label of samples (if other than 1)

log_min [float (default=0.001)] Minimum logarithm value for estimating AUC

log_max [float (default=1.)] Maximum logarithm value for estimating AUC.

Returns **auc** : float

semi-log ROC AUC

oddt.metrics.**enrichment_factor**(*y_true*, *y_score*, *percentage=1*, *pos_label=None*)

Computes enrichment factor for given percentage, i.e. EF_1% is enrichment factor for first percent of given samples.

Parameters **y_true** : array, shape=[n_samples]

True binary labels, in range {0,1} or {-1,1}. If positive label is different than 1, it must be explicitly defined.

y_score [array, shape=[n_samples]] Scores for tested series of samples

percentage [int or float] The percentage for which EF is being calculated

pos_label: int Positive label of samples (if other than 1)

Returns **ef** : float

Enrichment Factor for given percenage in range 0:1

oddt.metrics.**random_roc_log_auc**(*log_min=0.001*, *log_max=1.0*)

Computes area under semi-log ROC for random distribution.

Parameters **log_min** : float (default=0.001)

Minimum logarithm value for estimating AUC

log_max [float (default=1.)] Maximum logarithm value for estimating AUC.

Returns **auc** : float

semi-log ROC AUC for random distribution

oddt.metrics.**rmse**(*y_true*, *y_pred*)

Compute Root Mean Squared Error (RMSE)

Parameters **y_true** : array-like of shape = [n_samples] or [n_samples, n_outputs]

Ground truth (correct) target values.

y_pred [array-like of shape = [n_samples] or [n_samples, n_outputs]] Estimated target values.

Returns **rmse** : float

A positive floating point value (the best value is 0.0).

3.1.5 oddt.spatial module

Spatial functions included in ODDT Mainly used by other modules, but can be accessed directly.

oddt.spatial.**angle**(*p1*, *p2*, *p3*)

Returns an angle from a series of 3 points (point #2 is centroid).Angle is returned in degrees.

Parameters **p1,p2,p3** : numpy arrays, shape = [n_points, n_dimensions]

Triplets of points in n-dimensional space, aligned in rows.

Returns **angles** : numpy array, shape = [n_points]

Series of angles in degrees

oddt.spatial.**angle_2v**(*v1*, *v2*)

Returns an angle from a series of 3 points (point #2 is centroid).Angle is returned in degrees.

Parameters **v1,v2** : numpy arrays, shape = [n_vectors, n_dimensions]

Pairs of vectors in n-dimensional space, aligned in rows.

Returns **angles** : numpy array, shape = [n_vectors]

Series of angles in degrees

oddt.spatial.**dihedral**(*p1*, *p2*, *p3*, *p4*)

Returns an dihedral angle from a series of 4 points. Dihedral is returned in degrees. Function distingishes clockwise and antyclockwise dihedrals.

Parameters **p1,p2,p3,p4** : numpy arrays, shape = [n_points, n_dimensions]

Quadruplets of points in n-dimensional space, aligned in rows.

Returns **angles** : numpy array, shape = [n_points]

Series of angles in degrees

oddt.spatial.**distance**(*XA*, *XB*, *metric='euclidean'*, *p=2*, *V=None*, *VI=None*, *w=None*)

Computes distance between each pair of the two collections of inputs.

The following are common calling conventions:

1. `Y = cdist(XA, XB, 'euclidean')`

   Computes the distance between m points using Euclidean distance (2-norm) as the distance metric between the points. The points are arranged as m n-dimensional row vectors in the matrix X.

2. `Y = cdist(XA, XB, 'minkowski', p)`

   Computes the distances using the Minkowski distance $||u-v||_p$ (p-norm) where $p \geq 1$.

3. `Y = cdist(XA, XB, 'cityblock')`

   Computes the city block or Manhattan distance between the points.

4. `Y = cdist(XA, XB, 'seuclidean', V=None)`

   Computes the standardized Euclidean distance. The standardized Euclidean distance between two n-vectors `u` and `v` is

   $$\sqrt{\sum (u_i - v_i)^2 / V[x_i]}.$$

   V is the variance vector; V[i] is the variance computed over all the i'th components of the points. If not passed, it is automatically computed.

5. `Y = cdist(XA, XB, 'sqeuclidean')`

   Computes the squared Euclidean distance $||u-v||_2^2$ between the vectors.

6. `Y = cdist(XA, XB, 'cosine')`

   Computes the cosine distance between vectors u and v,

   $$1 - \frac{u \cdot v}{||u||_2 ||v||_2}$$

   where $||*||_2$ is the 2-norm of its argument $*$, and $u \cdot v$ is the dot product of u and v.

7. `Y = cdist(XA, XB, 'correlation')`

   Computes the correlation distance between vectors u and v. This is

   $$1 - \frac{(u - \bar{u}) \cdot (v - \bar{v})}{||(u - \bar{u})||_2 ||(v - \bar{v})||_2}$$

   where $\bar{v}$ is the mean of the elements of vector v, and $x \cdot y$ is the dot product of x and y.

8. `Y = cdist(XA, XB, 'hamming')`

   Computes the normalized Hamming distance, or the proportion of those vector elements between two n-vectors `u` and `v` which disagree. To save memory, the matrix `X` can be of type boolean.

9. `Y = cdist(XA, XB, 'jaccard')`

   Computes the Jaccard distance between the points. Given two vectors, `u` and `v`, the Jaccard distance is the proportion of those elements `u[i]` and `v[i]` that disagree where at least one of them is non-zero.

10. `Y = cdist(XA, XB, 'chebyshev')`

    Computes the Chebyshev distance between the points. The Chebyshev distance between two n-vectors `u` and `v` is the maximum norm-1 distance between their respective elements. More precisely, the distance is given by

    $$d(u, v) = \max_i |u_i - v_i|.$$

11. `Y = cdist(XA, XB, 'canberra')`

    Computes the Canberra distance between the points. The Canberra distance between two points `u` and `v` is

    $$d(u, v) = \sum_i \frac{|u_i - v_i|}{|u_i| + |v_i|}.$$

12. `Y = cdist(XA, XB, 'braycurtis')`

    Computes the Bray-Curtis distance between the points. The Bray-Curtis distance between two points `u` and `v` is

    $$d(u, v) = \frac{\sum_i (u_i - v_i)}{\sum_i (u_i + v_i)}$$

13. `Y = cdist(XA, XB, 'mahalanobis', VI=None)`

    Computes the Mahalanobis distance between the points. The Mahalanobis distance between two points `u` and `v` is $(u - v)(1/V)(u - v)^T$ where $(1/V)$ (the `VI` variable) is the inverse covariance. If `VI` is not None, `VI` will be used as the inverse covariance matrix.

14. `Y = cdist(XA, XB, 'yule')`

    Computes the Yule distance between the boolean vectors. (see yule function documentation)

15. `Y = cdist(XA, XB, 'matching')`

    Computes the matching distance between the boolean vectors. (see matching function documentation)

16. `Y = cdist(XA, XB, 'dice')`

    Computes the Dice distance between the boolean vectors. (see dice function documentation)

17. `Y = cdist(XA, XB, 'kulsinski')`

    Computes the Kulsinski distance between the boolean vectors. (see kulsinski function documentation)

18. `Y = cdist(XA, XB, 'rogerstanimoto')`

    Computes the Rogers-Tanimoto distance between the boolean vectors. (see rogerstanimoto function documentation)

19. `Y = cdist(XA, XB, 'russellrao')`

    Computes the Russell-Rao distance between the boolean vectors. (see russellrao function documentation)

20. `Y = cdist(XA, XB, 'sokalmichener')`

    Computes the Sokal-Michener distance between the boolean vectors. (see sokalmichener function documentation)

21. `Y = cdist(XA, XB, 'sokalsneath')`

    Computes the Sokal-Sneath distance between the vectors. (see sokalsneath function documentation)

22. `Y = cdist(XA, XB, 'wminkowski')`

    Computes the weighted Minkowski distance between the vectors. (see sokalsneath function documentation)

23. `Y = cdist(XA, XB, f)`

    Computes the distance between all pairs of vectors in X using the user supplied 2-arity function f. For example, Euclidean distance between the vectors could be computed as follows:

```
dm = cdist(XA, XB, lambda u, v: np.sqrt(((u-v)**2).sum()))
```

    Note that you should avoid passing a reference to one of the distance functions defined in this library. For example,:

```
dm = cdist(XA, XB, sokalsneath)
```

    would calculate the pair-wise distances between the vectors in X using the Python function sokalsneath. This would result in sokalsneath being called $\binom{n}{2}$ times, which is inefficient. Instead, the optimized C version is more efficient, and we call it using the following syntax.:

```
dm = cdist(XA, XB, 'sokalsneath')
```

Parameters **XA** : ndarray

An m_A by n array of m_A original observations in an n-dimensional space.

XB : ndarray

An m_B by n array of m_B original observations in an n-dimensional space.

metric : string or function

The distance metric to use. The distance function can be 'braycurtis', 'canberra', 'chebyshev', 'cityblock', 'correlation', 'cosine', 'dice', 'euclidean', 'hamming', 'jaccard', 'kulsinski', 'mahalanobis', 'matching', 'minkowski', 'rogerstanimoto', 'russellrao', 'seuclidean', 'sokalmichener', 'sokalsneath', 'sqeuclidean', 'wminkowski', 'yule'.

w : ndarray

The weight vector (for weighted Minkowski).

p : double

The p-norm to apply (for Minkowski, weighted and unweighted)

V : ndarray

The variance vector (for standardized Euclidean).

VI : ndarray

The inverse of the covariance matrix (for Mahalanobis).

Returns **Y** : ndarray

A m_A by m_B distance matrix is returned. For each i and j, the metric `dist(u=XA[i], v=XB[j])` is computed and stored in the ij th entry.

Raises **An exception is thrown if ''XA'' and ''XB'' do not have**

the same number of columns.

3.1.6 oddt.virtualscreening module

ODDT pipeline framework for virtual screening

class `oddt.virtualscreening.`**`virtualscreening`**(*n_cpu=-1*, *verbose=False*)

Virtual Screening pipeline stack

Parameters **n_cpu: int (default=-1)**

The number of parallel procesors to use

verbose: bool (default=False) Verbosity flag for some methods

Methods

apply_filter(expression[, filter_type, ...])	Filtering method, can use raw expressions (strings to be evaled in if statement, can u
dock(engine, protein, *args, **kwargs)	Docking procedure.
fetch()	
load_ligands(file_type, ligands_file)	Loads file with ligands.
score(function, protein, *args, **kwargs)	Scoring procedure.
write(fmt, filename[, csv_filename])	Outputs molecules to a file
write_csv(csv_filename[, keep_pipe])	Outputs molecules to a csv file

apply_filter(*expression*, *filter_type='expression'*, *soft_fail=0*)
Filtering method, can use raw expressions (strings to be evaled in if statement, can use oddt.toolkit.Molecule methods, eg. 'mol.molwt < 500') Currently supported presets:

•Lipinski Rule of 5 ('r5' or 'l5')

•Fragment Rule of 3 ('r3')

Parameters **expression: string or list of strings**

Expresion(s) to be used while filtering.

filter_type: 'expression' or 'preset' (default='expression') Specify filter type: 'expression' or 'preset'. Default strings are treated as expressions.

soft_fail: int (default=0) The number of faulures molecule can have to pass filter, aka. soft-fails.

dock(*engine*, *protein*, **args*, ***kwargs*)
Docking procedure.

Parameters **engine: string**

Which docking engine to use.

fetch()

load_ligands(*file_type*, *ligands_file*)
Loads file with ligands.

Parameters **file_type: string**

Type of molecular file

ligands_file: string Path to a file, which is loaded to pipeline

score(*function*, *protein*, **args*, ***kwargs*)
Scoring procedure.

Parameters **function: string**

Which scoring function to use.

protein: oddt.toolkit.Molecule Default protein to use as reference

write(*fmt*, *filename*, *csv_filename=None*, ***kwargs*)
Outputs molecules to a file

Parameters **file_type: string**

Type of molecular file

ligands_file: string Path to a output file

csv_filename: string Optional path to a CSV file

write_csv(*csv_filename*, *keep_pipe=False*, ***kwargs*)
Outputs molecules to a csv file

Parameters **csv_filename: string**

Optional path to a CSV file

keep_pipe: bool (default=False) If set to True, the ligand pipe is sustained.

3.1.7 Module contents

Open Drug Discovery Toolkit

Universal and easy to use resource for various drug discovery tasks, ie docking, virutal screening, rescoring.

toolkit [module,] Toolkits backend module, currenlty OpenBabel [ob] and RDKit [rdk]. This setting is toolkit-wide, and sets given toolkit as default

CHAPTER 4

References

To be announced.

CHAPTER 5

Docuimentation Indices and tables

- genindex
- modindex
- search

[R1] Wikipedia entry for the Receiver operating characteristic

[R2] Wikipedia entry for the Receiver operating characteristic

O

A

B

C

D

E

F

H

U

V

W