MLZ Documentation

Release 1.0

Matias Carrasco Kind

January 15, 2014

CONTENTS

About

MLZ, "**M**achine **L**earning and photo-**Z**" is a parallel python framework that computes fast and robust photometric redshift PDFs using Machine Learning algorithms. In particular, it uses a supervised technique with prediction trees and random forest through *TPZ* or a unsupervised methods with self organizing maps and random atlas through *SOMz*. It can be easily extended to other regression or classification problems.

CHAPTER

ONE

REFERENCES

These are the references related to this framework where detailed information about these methods can be found.

- Carrasco Kind, M., & Brunner, R. J., 2013 "TPZ : Photometric redshift PDFs and ancillary information by using prediction trees and random forests", MNRAS, 432, 1483 (Link)
- Carrasco Kind, M., & Brunner, R. J., 2014, "SOMz : photometric redshift PDFs with self organizing maps and random atlas" , MNRAS, in press. (Link)
- Carrasco Kind, M., & Brunner, R. J., 2013, "Implementing Probabilistic Photometric Redshifts", in Astronomical Society of the Pacific Conference Series, Vol. 475, Astronomical Data Analysis Software and Systems XXII (ADASSXXII), Freidel D., ed., p. 69 (Link)

CHAPTER

TWO

CONTACT

Here you can find my contact information for questions or comments.

CHAPTER

THREE

CONTENTS

This is a brief documentation of MLZ and the routines included

3.1 Requirements

The standard requirements for MLZ are the following python libraries:

- numpy
- scipy
- matplotlib
- mpi4py (for parallel running)
- healpy (for spherical coordinates) **optional but recommended**
- f2py (a fortran wrapper) **optional but recommended**

3.1.1 Parallel MLZ

MLZ can run on a single node but it is **strongly** recommended to install MPI libraries and mpi4py.

In order to run in parallel MPI libraries must be present and the mpi4py module which can be obtained from http://mpi4py.scipy.org/ . A first you might get it easy by trying:

```
[sudo] easy_install install mpi4py
```

ot via pip:

```
[sudo] pip install mpi4py
```

with sudo permissions, or locally using:

```
easy_install --user install mpi4py
```

or:

```
pip install mpi4py --user
```

3.2 Installation

3.2.1 Download

You can get the latest version of the code from: https://pypi.python.org/pypi/MLZ/1.0

The simplest way to get MLZ is using pip:

```
[sudo] pip install MLZ
```

or:

```
pip install MLZ --user
```

or even:

```
easy_install --user MLZ
```

which will install and copy all the files to a local directory which can be specified by using the `--prefix=<path>` flag. Open python and check if the module is present:

```
>>> import mlz
```

The code can also be installed manually, by getting the tar file and then uncompress the file:

```
tar -zxf MLZ-1.0.tar.gz
```

and install using:

```
python setup.py install --user
```

which is not required for the code to run via standard way. This will create a copy of the code in a local directory and a few executable files will be copied to a local script folder (usually at $HOME/.local/bin). This also compiles automatically the fortran routines needed for better performance.

3.2.2 Fortran routines

If the installation did not work via `setup.py` as shown, some fortran libraries will not be present, these can be manually compiled using the f2py wrapper. In the mlz folder, go to the ml_codes folder where you will find the file `som.f90` in that folder:

```
f2py -c -m somF som.f90
```

And the library `somF.so` will be created. The code still works even this step is not accomplished as these routines aid the code to run more efficiently.

3.2.3 Release Note

MLZ is an open source code released and *licensed* under the University of Illinois/NCSA Open Source License and it is distributed *without any warranty*.

3.2.4 Acknowledgement

Please, acknowledge the use of MLZ in your own work with this (or similar) with these *references*

3.2.5 Uninstall

You can uninstall MLZ by deleting the files manually with:

```
python setup.py install --user --record installed_files.txt
cat installed_files.txt | xargs rm -rf
rm -rf installed_files
```

Then proceed to delete mlz folder in the local installation.

Or you can use:

```
pip uninstall MLZ
```

3.3 Machine Learning routines

MLZ uses two methods to compute, primarily, photometric redshift PDFs. It uses a supervised technique called TPZ [1] which uses prediction trees and random forest methods to make predictions in a regression or classification problem. We also have implemented a unsupervised methods using self organizing maps and introducing random atlas called SOMz [2].

Both method can be called from the main routine to obtain results from different points of view, we are currently working on how efficiently combine these and other methods taking advantage of their strengths.

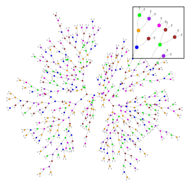
The methods included are the following:

3.3.1 TPZ: Trees for Photo-Z

TPZ [3] is a supervised machine learning, parallel algorithm that uses prediction trees and random forest techniques to produce both robust photometric redshift PDFs and ancillary information for a galaxy sample. A prediction tree is built by asking a sequence of questions that recursively split the input data taken from the spectroscopic sample, frequently into two branches, until a terminal leaf is created that meets a stopping criterion (e.g., a minimum leaf size or a variance threshold).

The dimension in which the data is divided is chosen to be the one with highest information gain among the random subsample of dimensions obtained at every point. This process produces less correlated trees and allows to explore several configurations within the data.

The small region bounding the data in the terminal leaf node represents a specific subsample of the entire data with similar properties. Within this leaf, a model is applied that provides a fairly comprehensible prediction, especially in situations where many variables may exist that interact in a nonlinear manner as is often the case with photo-z estimation.



[1] Carrasco Kind, M., & Brunner, R. J., 2013 “TPZ : Photometric redshift PDFs and ancillary information by using prediction trees and random forests”, MNRAS, 432, 1483 (Link)

[2] Carrasco Kind, M., & Brunner, R. J., 2014, “SOMz : photometric redshift PDFs with self organizing maps and random atlas” , MNRAS, in press. (Link)

[3] Carrasco Kind, M., & Brunner, R. J., 2013 “TPZ : Photometric redshift PDFs and ancillary information by using prediction trees and random forests”, MNRAS, 432, 1483 (Link)

In the code TPZ is implemented as a module which has 2 important classes: `TPZ.Rtree` for regression and `TPZ.Ctree` for classification. Both are documented in the code and listed below. For more information please refer to the TPZ paper

Regression Tree Class

This is the `TPZ.Rtree` class in some detail, refer to the source code for mode information and methods.

Module author: Matias Carrasco Kind

class `TPZ.`**`Rtree`**(*X*, *Y*, *minleaf=4*, *forest='yes'*, *mstar=2*, *dict_dim='*')

Creates a regression tree class instance

Parameters

- **X** (*float or int array, 1 row per object*) – Preprocessed attributes array (*all* columns are considered)
- **Y** (*float*) – Attribute to be predicted
- **minleaf** (*int, def = 4*) – Minimum number of objects on terminal leaf
- **forest** (*str, 'yes'/'no'*) – Random forest key
- **mstar** (*int*) – Number of random subsample of attributes if forest is used
- **dict_dim** (*dict*) – dictionary with attributes names

`get_branch`(*line*)

Get the branch in string format given a line search, where the line is a vector of attributes per individual object

Parameters **line** (*float*) – input data line to look in the tree, same dimensions as input X

Returns str – branch array in string format, ex., ['L','L','R']

`get_vals`(*line*)

Get the predictions given a line search, where the line is a vector of attributes per individual object

Parameters **line** (*float*) – input data line to look in the tree, same dimensions as input X

Returns float – array with the leaf content

`leaves`()

Return an array with all branches in string format ex: ['L','R','L'] is a branch of depth 3 where L and R are the left or right branches

Returns str – Array of all branches in the tree

`leaves_dim`()

Returns an array of the used dimensions for all the the nodes on all the branches

Returns int – Array of all the dimensions for each node on each branch

`plot_tree`(*itn=-1*, *fileout='TPZ'*, *path='*', *save_png='no'*)

Plot a tree using dot (Graphviz) Saves it into a png file by default

Parameters

- **itn** (*int*) – Number of tree to be included on path, use -1 to ignore this number
- **fileout** (*str*) – Name of file for the png files
- **path** (*str*) – path for the output files
- **save_png** (*str*) – save png created by Graphviz ('yes'/'no')

`print_branch`(*branch*)
Returns the content of a leaf on a branch (given in string format)

`save_tree`(*itn=-1*, *fileout='TPZ'*, *path=''*)
Saves the tree

Parameters

- **itn** (*int*) – Number of tree to be included on path, use -1 to ignore this number
- **fileout** (*str*) – Name of output file
- **path** (*str*) – path for the output file

Classification Trees Class

This is the `TPZ.Ctree` class in some detail, refer to the source code for mode information and methods.

Module author: Matias Carrasco Kind

class `TPZ.`**`Ctree`**(*X*, *Y*, *minleaf=4*, *forest='yes'*, *mstar=2*, *dict_dim=''*, *impurity='entropy'*, *nclass=array([0, 1])*)
Creates a classification tree class instance

Parameters

- **X** (*float or int array, 1 row per object*) – Preprocessed attributes array (*all* columns are considered)
- **Y** (*int array*) – Attribute to be predicted
- **minleaf** (*int, def = 4*) – Minimum number of objects on terminal leaf
- **forest** (*str, 'yes'/'no'*) – Random forest key
- **mstar** (*int*) – Number of random subsample of attributes if forest is used
- **impurity** – 'entropy'/'gini'/'classE' to compute information gain
- **nclass** (*int array*) – classes array (labels)
- **dict_dim** (*dict*) – dictionary with attributes names

`get_branch`(*line*)
Same as `Rtree.get_branch()`

`get_vals`(*line*)
Same as `Rtree.get_vals()`

`leaves`()
Same as `Rtree.leaves()`

`leaves_dim`()
Same as `Rtree.leaves_dim()`

`plot_tree`(*itn=-1*, *fileout='TPZ'*, *path=''*, *save_png='no'*)
Same as `Rtree.plot_tree()`

`print_branch`(*branch*)
Same as `Rtree.print_branch()`

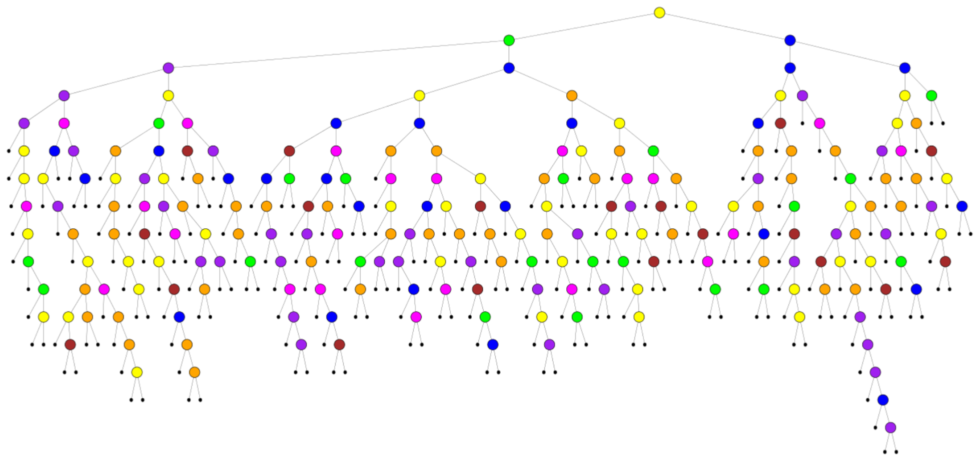
`save_tree`(*itn=-1*, *fileout='TPZ'*, *path=''*)
Same as `Rtree.save_tree()`

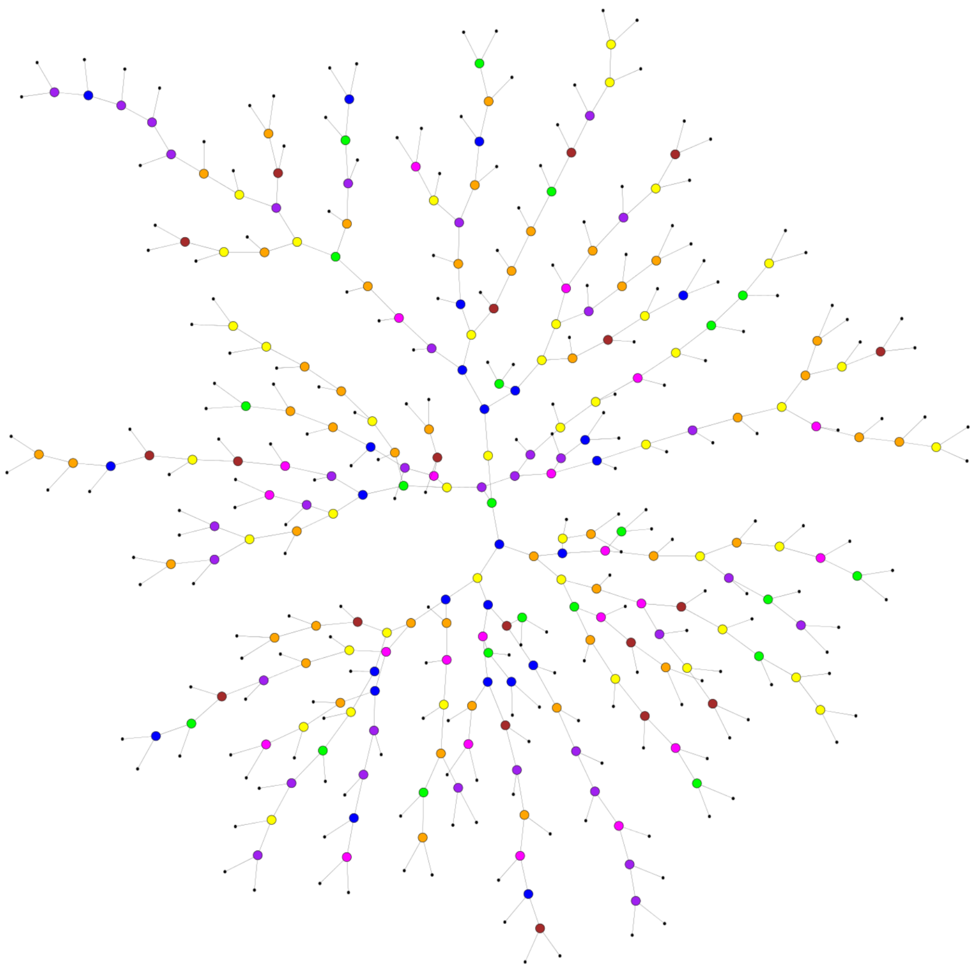
Warning: In order to visualize the created trees you need to have installed Graphviz, usually is installed by default on Linux and Mac OS systems You don't needed it in order to run MLZ

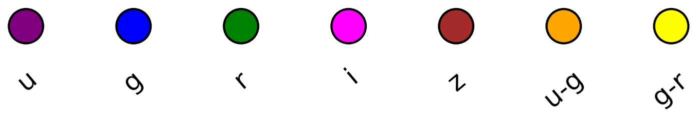
Example 1

This is a simple example on how to use the `TPZ.Rtree`, visualize a tree and make a simple prediction. To see an example of using this properly in a problem under the MLZ framework , see *Running a test*

```
from numpy import *
import os, sys

path_src = os.path.abspath(os.path.join(os.getcwd(), '../../'))
if not path_src in sys.path: sys.path.insert(1, path_src)
from mlz.ml_codes import *

#X and Y can be anything, in this case SDSS mags and colors for X and photo-z for Y
X = loadtxt('SDSS_MGS.train', usecols=(1, 2, 3, 4, 5, 6, 7), unpack=True).T
Y = loadtxt('SDSS_MGS.train', unpack=True, usecols=(0,))

#this dictionary is optional for this example
#for plotting the color labels
#(automatically included in MLZ)
d = {'u': {'ind': 0}, 'g': {'ind': 1}, 'r': {'ind': 2}, 'i': {'ind': 3}, 'z': {'ind': 4}, 'u-g': {'i
     'g-r': {'ind': 6}}

#Calls the Regression Tree mode
T = TPZ.Rtree(X, Y, minleaf=30, mstar=3, dict_dim=d)
T.plot_tree()
#get a list of all branches
branches = T.leaves()
#print first branch, in this case left ,left, left, etc...
print 'branch = ', branches[0]
#print content of branch
content = T.print_branch(branches[0])
print 'branch content'
print content
#get prediction values for a test data (just an example on how to do it)
#using a train objetc
values = T.get_vals(X[10])
print 'predicted values from tree'
print values
print
print 'mean value from prediction', mean(values)
print 'real value', Y[10]
#Note we use a shallow tree and only one tree for example purposes and there
#is a random subsmaple so answer changes every time
```







If you `download` this example and run it on a python console you would get the following output, although the final line would change slightly as there is a random process involved which would also change the figures:

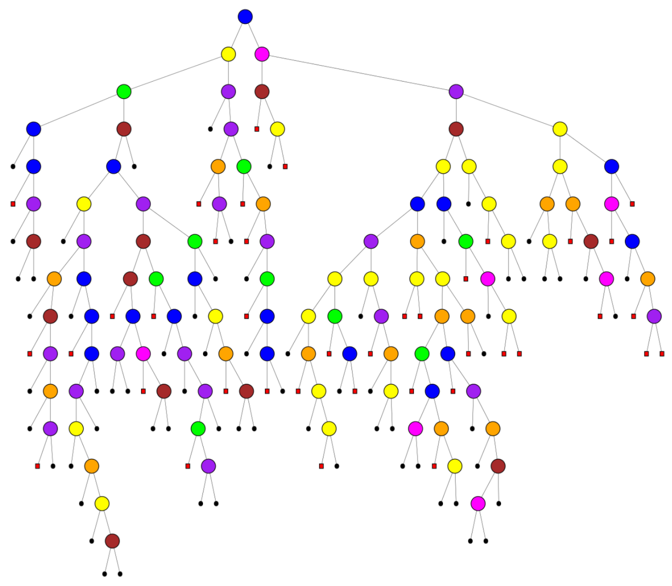
```
>>> branches = T.leaves()
>>> print 'branch = ', branches[0]
branch =  ['L', 'L', 'L', 'L', 'L']
>>> content = T.print_branch(branches[0])
>>> print 'brach content'
branch content
>>> print content
[ 0.024914  0.029343  0.005126  0.017902  0.019716  0.02609   0.004404
  0.006451  0.003074  0.034597  0.005701  0.003923  0.032468  0.031017
  0.023015  0.038875  0.010996  0.018425  0.007773  0.013524  0.024911
  0.003017  0.013113  0.006682  0.007372  0.021268]
>>> values = T.get_vals(X[10])
>>> print 'predicted values from tree'
>>> print values
[ 0.120684  0.118015  0.108008  0.103931  0.11477   0.099268  0.106299
  0.114634  0.11031   0.115252  0.102601  0.132789  0.12069   0.125127
  0.115067  0.086241  0.115476  0.112288  0.096661  0.105071  0.108449
  0.119887  0.111333  0.120343  0.130859  0.104452  0.126068  0.095225
  0.102079  0.123717  0.118518  0.116976  0.094429  0.107744  0.111157
  0.095198  0.127612  0.114376  0.105994  0.117298  0.105951  0.09058
  0.118837  0.108803  0.114075  0.159866  0.116929  0.086987  0.099276
  0.088263  0.117582  0.119883  0.126069  0.117097  0.110187  0.099429
  0.102188  0.105896  0.107781]
>>> print 'mean value from prediction', mean(values)
mean value from prediction 0.111365677966
>>> print 'real value', Y[10]
real value 0.120684
```

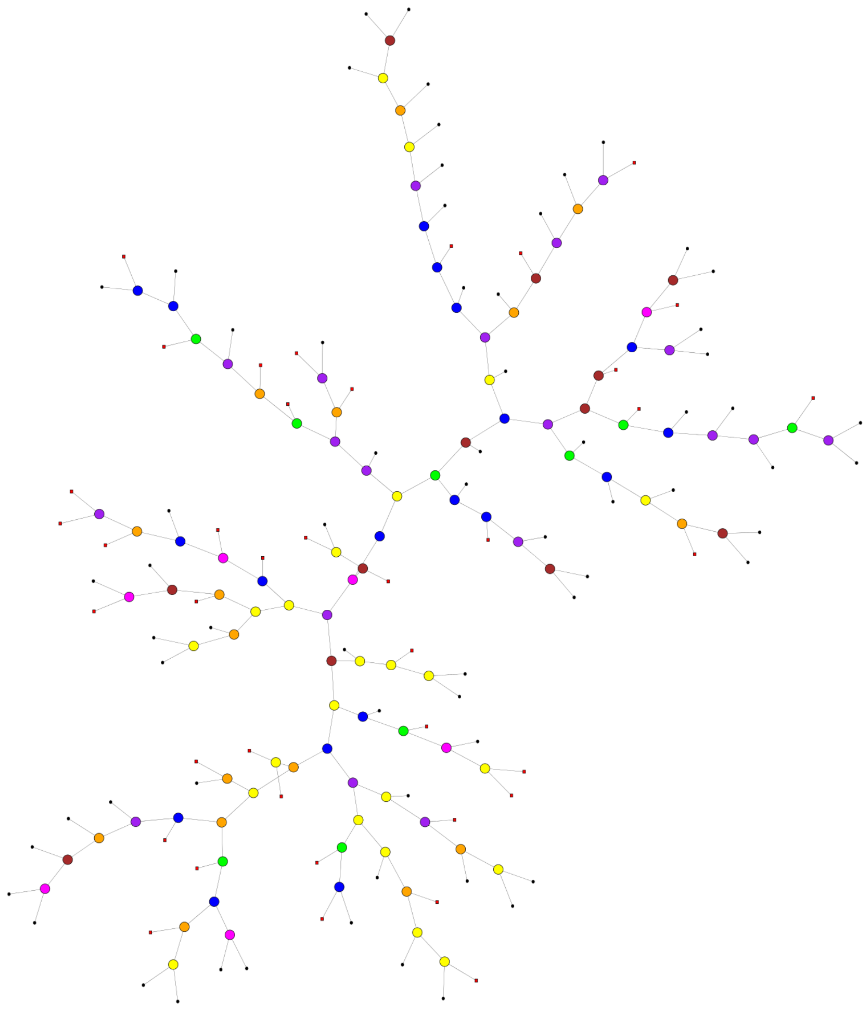
Example 2

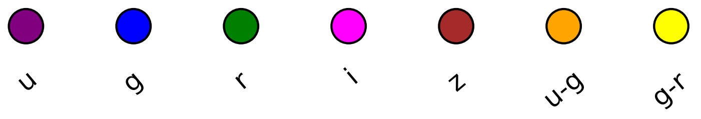
This is a simple `example` on how to use the `TPZ.Ctree`, to visualize a tree and to make a simple classification, in this case we classify from low and high redshift. Note the differences with these tree as leaf are painted according to different classes.

```
from numpy import *
import os, sys

path_src = os.path.abspath(os.path.join(os.getcwd(), '../../'))
if not path_src in sys.path: sys.path.insert(1, path_src)
from mlz.ml_codes import *

#X and Y can be anything, in this case SDSS mags and colors for X and photo-z for Y
X = loadtxt('SDSS_MGS.train', usecols=(1, 2, 3, 4, 5, 6, 7), unpack=True).T
Y = loadtxt('SDSS_MGS.train', unpack=True, usecols=(0,))

#make two classes by separating Y in low and high redhisft for example

Y = where((Y > 0.15), 1, 0)

#0: low redshift, 1: high redshift

#this dictionary is optional for this example
#for plotting the color labels
#(automatically included in MLZ)
d = {'u': {'ind': 0}, 'g': {'ind': 1}, 'r': {'ind': 2}, 'i': {'ind': 3}, 'z': {'ind': 4}, 'u-g': {'in
     'g-r': {'ind': 6}}
```

```
#Calls the Classification Tree mode
T = TPZ.Ctree(X, Y, minleaf=20, mstar=3, dict_dim=d, nclass=array([0, 1], dtype='int'))
T.plot_tree()
#get a list of all branches
branches = T.leaves()
#print first branch, in this case left ,left, left, etc...
print 'branch = ', branches[0]
#print content of branch
content = T.print_branch(branches[0])
print 'branch content'
print content
#get prediction values for a test data (just an example on how to do it)
#using a train objetc
values = T.get_vals(X[20])
print 'predicted values from tree'
print values
print
print 'mean value from prediction', int(round(mean(values)))
print 'real value', Y[20]
#Note we use a shallow tree and only one tree for example purposes and there
#is a random subsmaple so answer changes every time
```







References

3.3.2 SOMz: Self Organizing Maps and random atlas

SOMz [4] is a unsupervised machine learning technique that also computes photometric redshift PDFs. Specifically, we have developed a new framework that we have named random atlas, which mimics the random forest approach by replacing the prediction trees with self organizing maps (SOMs). A SOM is essentially a neural network that maps a large training set via a process of competitive learning from a high dimensional input space to a two-dimensional surface. The mapping process retains the topology of the input data, thereby revealing potential unknown correlations between input parameters, which can provide important insights into the data.

This is an unsupervised learning method as no prediction attributes are included in the mapping process, only the non-prediction attributes are included. The output values from the training data are only used after the map has been constructed as they can be used to generate the prediction model for each cell in the map. In our implementation , we first construct a suite of maps that each use a random subset of the full attributes and the randomized training data we developed for the random forest, and we then aggregate the map predictions together to make our final prediction (via the random atlas).

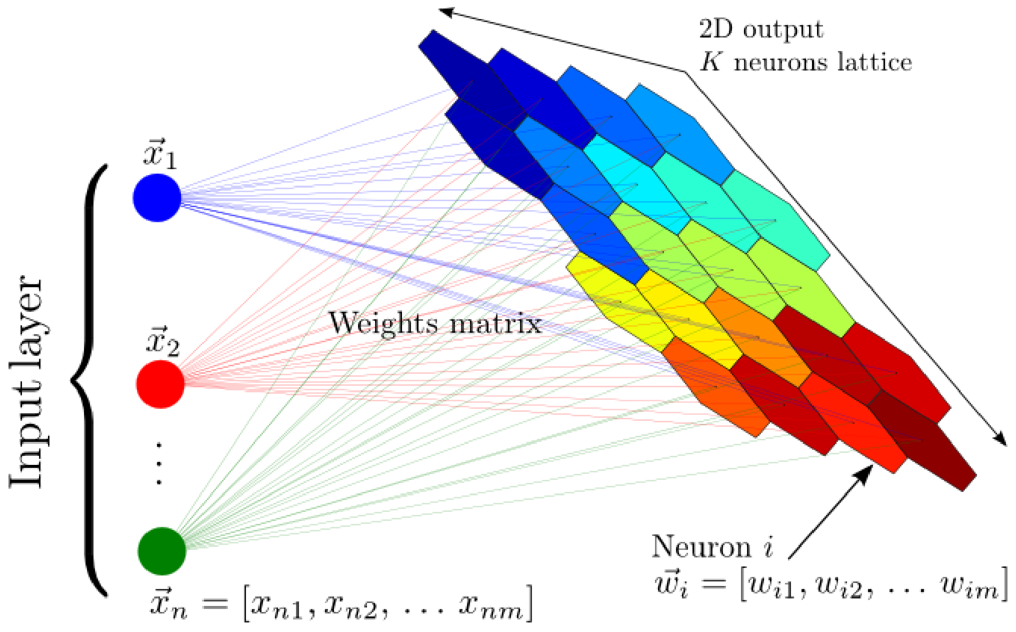

2D output
K neurons lattice
$\vec{x}_1$
$\vec{x}_2$
Input layer
Weights matrix
Neuron i
$\vec{w}_i = [w_{i1}, w_{i2}, \ldots w_{im}]$
$\vec{x}_n = [x_{n1}, x_{n2}, \ldots x_{nm}]$

Figure 3.1: A schematic representation of a self organized map. The color of the map encodes the organization of groups of objects with similar properties. The main feature of the SOM is that produces a nonlinear mapping from a m-dimensional space of attributes to a two-dimensional lattices of neurons

In the code SOMz is implemented as a module `SOMZ` to create, evaluate, plot and make prediction. Given the nature of the algorithm this can also be used for both, regression and classifcatin it just a matter of changing the attributes when evaluating. For more details refer to the SOMz paper

Somz module

This is the `SOMZ` class in some detail, refer to the source code for mode information and methods.

Module author: Matias Carrasco Kind

[4] Carrasco Kind, M., & Brunner, R. J., 2014, “SOMz : photometric redshift PDFs with self organizing maps and random atlas” , MNRAS, in press. (Link)

class `SOMZ.`**`SelfMap`**(*X*, *Y*, *topology='grid'*, *som_type='online'*, *Ntop=28*, *iterations=30*, *periodic='no'*, *dict_dim=''*, *astart=0.8*, *aend=0.5*, *importance=None*)

Create a som class instance

Parameters

- **X** (*float*) – Attributes array (all columns used)
- **Y** (*float*) – Attribute to be predicted (not really needed, can be zeros)
- **topology** (*str*) – Which 2D topology, 'grid', 'hex' or 'sphere'
- **som_type** (*str*) – Which updating scheme to use 'online' or 'batch'
- **Ntop** (*int*) – Size of map, for grid Size=Ntop*Ntop, for hex Size=Ntop*(Ntop+1[2]) if Ntop is even[odd] and for sphere Size=12*Ntop*Ntop and top must be power of 2
- **iterations** (*int*) – Number of iteration the entire sample is processed
- **periodic** (*str*) – Use periodic boundary conditions ('yes'/'no'), valid for 'hex' and 'grid' only
- **dict_dim** (*dict*) – dictionary with attributes names
- **astar** (*float*) – Initial value of alpha
- **aend** (*float*) – End value of alpha
- **importance** (*str*) – Path to the file with importance ranking for attributes, default is none

`create_map`(*evol='no'*, *inputs_weights=''*)

This is same as above but uses python routines instead

`create_mapF`(*evol='no'*, *inputs_weights=''*)

This functions actually create the maps, it uses random values to initialize the weights It uses a Fortran subroutine compiled with f2py

`evaluate_map`(*inputX=''*, *inputY=''*)

This functions evaluates the map created using the input Y or a new Y (array of labeled attributes) It uses the X array passed or new data X as well, the map doesn't change

Parameters

- **inputX** (*float*) – Use this if another set of values for X is wanted using the weigths already computed
- **inputY** (*float*) – One dimensional array of the values to be assigned to each cell in the map based on the in-memory X passed

`get_best`(*line*)

Get the predictions given a line search, where the line is a vector of attributes per individual object for THE best cell

Parameters **line** (*float*) – input data to look in the tree

Returns array with the cell content

`get_vals`(*line*)

Get the predictions given a line search, where the line is a vector of attributes per individual object fot the 10 closest cells.

Parameters **line** (*float*) – input data to look in the tree

Returns array with the cell content

`plot_map`(*min_m=-100*, *max_m=-100*, *colbar='yes'*)

Plots the map after evaluating, the cells are colored with the mean value inside each one of them

Parameters

- **min_m** (*float*) – Lower limit for coloring the cells, -100 uses min value
- **max_m** (*float*) – Upper limit for coloring the cells, -100 uses max value
- **colbar** (*str*) – Include a colorbar ('yes','no')

`save_map`(*itn=-1*, *fileout='SOM'*, *path=''*)
Saves the map

Parameters

- **itn** (*int*) – Number of map to be included on path, use -1 to ignore this number
- **fileout** (*str*) – Name of output file
- **path** (*str*) – path for the output file

`save_map_dict`(*path=''*, *fileout='SOM'*, *itn=-1*)
Saves the map in dictionary format

Parameters

- **itn** (*int*) – Number of map to be included on path, use -1 to ignore this number
- **fileout** (*str*) – Name of output file
- **path** (*str*) – path for the output file

`som_best_cell`(*inputs*, *return_vals=1*)
Return the closest cell to the input object It can return more than one value if needed

`SOMZ.geometry`(*top*, *Ntop*, *periodic='no'*)
Pre-compute distances between cells in a given topology and store it on a distLib array

Parameters

- **top** (*str*) – Topology ('grid','hex','sphere')
- **Ntop** (*int*) – Size of map, for grid Size=Ntop*Ntop, for hex Size=Ntop*(Ntop+1[2]) if Ntop is even[odd] and for sphere Size=12*Ntop*Ntop and top must be power of 2
- **periodic** (*str*) – Use periodic boundary conditions ('yes'/'no'), valid for 'hex' and 'grid' only

Returns 2D array with distances pre computed between cells and total number of units

Return type 2D float array, int

`SOMZ.get_alpha`(*t*, *alphas*, *alphae*, *NT*)
Get value of alpha at a given time

`SOMZ.get_ns`(*ix*, *iy*, *nx*, *ny*, *index=False*)
Get neighbors for rectangular grid given its coordinates and size of grid

Parameters

- **ix** (*int*) – Coordinate in the x-axis
- **iy** (*int*) – Coordinate in the y-axis
- **nx** (*int*) – Number fo cells along the x-axis
- **ny** (*int*) – Number fo cells along the y-axis
- **index** (*bool*) – Return indexes in the map format

Returns Array of indexes for direct neighbors

SOMZ.**get_ns_hex**(*ix*, *iy*, *nx*, *ny*, *index=False*)
Get neighbors for hexagonal grid given its coordinates and size of grid Same parameters as `get_ns()`

SOMZ.**get_sigma**(*t*, *sigma0*, *sigmaf*, *NT*)
Get value of sigma at a given time

SOMZ.**h**(*bmu*, *mapD*, *sigma*)
Neighborhood function which quantifies how much cells around the best matching one are modified

Parameters

- **bmu** (*int*) – best matching unit
- **mapD** (*float*) – array of distances computed with `geometry()`

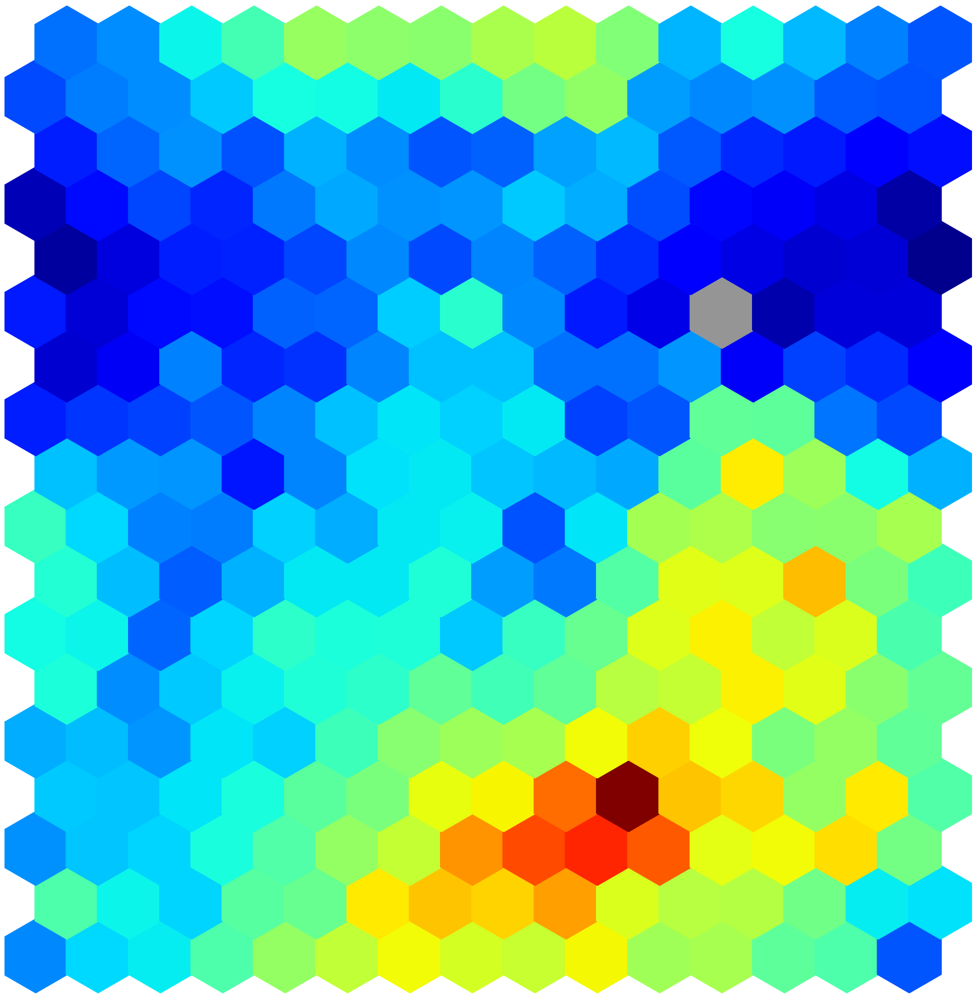
SOMZ.**is_power_2**(*value*)
Check if passed value is a power of 2

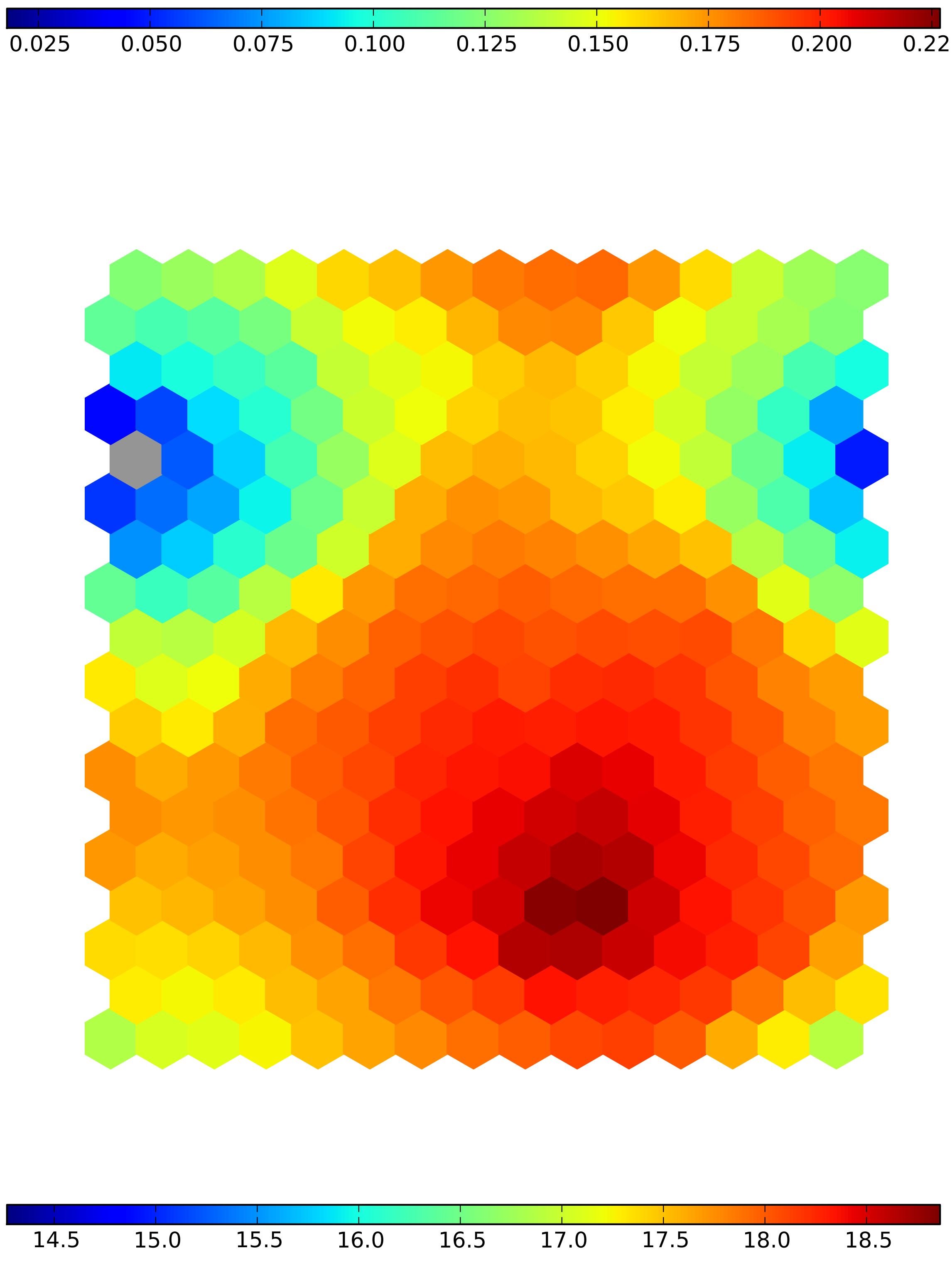
Example

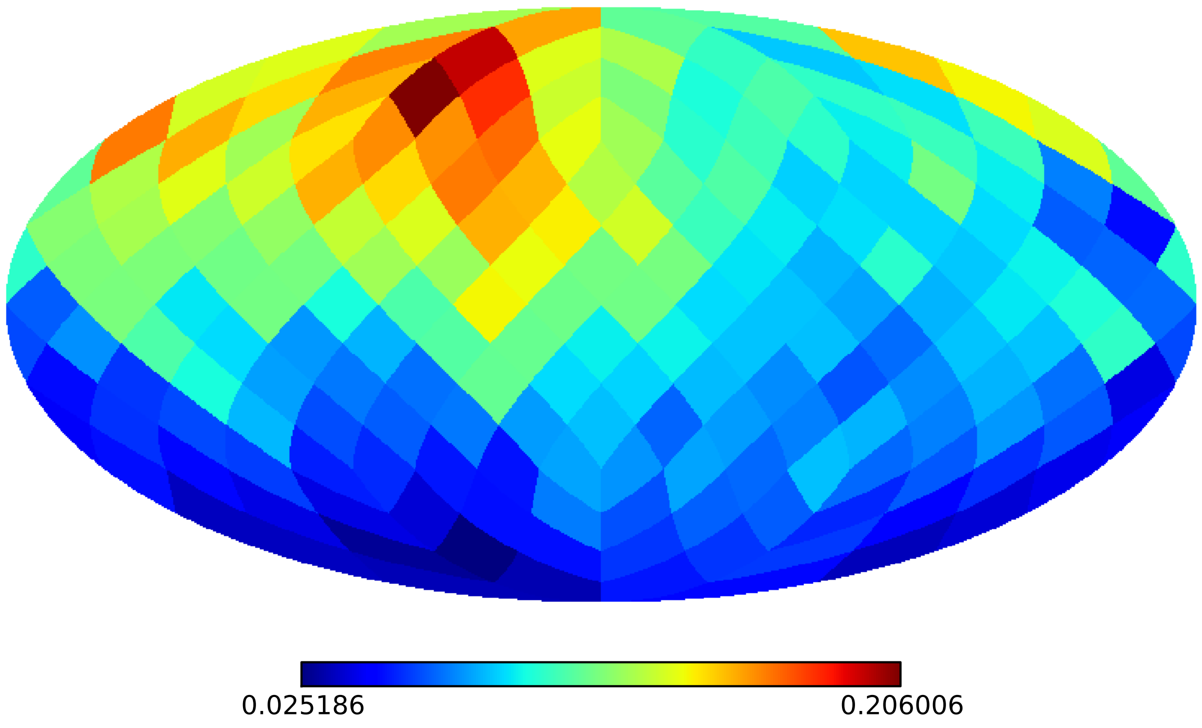
This is a simple example on how to use the `SOMZ`, visualize a map and make a simple prediction. To see an example of using this properly in a problem under the MLZ framework , see *Running a test*

```
from numpy import *
import os, sys

path_src = os.path.abspath(os.path.join(os.getcwd(), '../../'))
if not path_src in sys.path: sys.path.insert(1, path_src)
from mlz.ml_codes import *

#X and Y can be anything, in this case SDSS mags and colors for X and photo-z for Y
X = loadtxt('SDSS_MGS.train', usecols=(1, 2, 3, 4, 5, 6, 7), unpack=True).T
Y = loadtxt('SDSS_MGS.train', unpack=True, usecols=(0,))


#Calls the SOMZ mode
M = SOMZ.SelfMap(X,Y,topology='hex',Ntop=15,iterations=100,periodic='yes')
#creates a map
M.create_mapF()
#evaluates it with the Y entered, or anyoher desired colum
M.evaluate_map()
#plots the map
M.plot_map()
#get prediction values for a test data (just an example on how to do it)
#using a train objetc
values = M.get_vals(X[10])
print
print 'mean value from prediction (hex)', mean(values)
print 'real value', Y[10]
#Note we use a low-resoution map and only one map for example purposes
#evaluate other column, for example the 'g' magnitude
M.evaluate_map(inputY=X[:,1])
M.plot_map()


#Try other topology
M = SOMZ.SelfMap(X,Y,topology='sphere',Ntop=4,iterations=100,periodic='yes')
#creates a map
M.create_mapF()
```

```
#evaluates it with the Y entered, or anyoher desired colum
M.evaluate_map()
#plots the map
M.plot_map()
#get prediction values for a test data (just an example on how to do it)
#using a train objetc
values = M.get_vals(X[10])
print
print 'mean value from prediction (sphere)', mean(values)
print 'real value', Y[10]
```




0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.200
0.225
14.5
15.0
15.5
16.0
16.5
17.0
17.5
18.0
18.5



You can `download` this example and run it on a python console to see the outputs and try different topologies, properties and resolutions.

References

References

3.4 Other routines

Here we list other modules used by MLZ that help the code to run efficiently. for a more detailed please refer to the source code, these are samples of some important routines used by the code. Check *Running a test* for an example on how to run MLZ for a SDSS sample of galaxies

3.4.1 Data module

This is the `data` class in some detail, this class is useful to read the data in a specified format and to read the inputs parameters as specified in the *input-file*.

Module author: Matias Carrasco Kind

`data.`**`bootstrap_index`**(*N*, *SS*)
Returns bootstrapping indexes of sample N from array of indices

Parameters

- **N** (*int*) – size of boostrap sample
- **SS** (*int*) – extract indexes from 0 to SS

Returns array of bootstrap indices

Return type int array

class `data.`**`catalog`**(*Pars*, *cat_type='train'*, *L1=0*, *L2=-1*, *rank=0*)
Creates a catalog instance for training or testing

Parameters

- **Pars** (*class*) – Class of parameters read from inputs files
- **cat_type** (*str*) – 'train' or 'test' file (names are taken from Pars class)
- **L1** (*int*) – keep only entries between L1 and L2
- **L2** (*int*) – keep only entries between L1 and L2

`get_XY`(*curr_at='all'*, *bootstrap='no'*)
Creates X and Y methods based on catalog, using random realization or bootstrapping, after this both X and Y are loaded and ready to be used

Parameters

- **curr_at** (*dict*) – dictionary of attributes to be used (like a subsample of them), 'all' by default
- **bootstrap** (*str*) – Bootstrapping sample? ('yes'/'no')

Returns Saves X, Y oob (and no-oob) data if required and original catalog

`has_X`()
Is X already loaded in memory?

Returns Boolean

`has_Y`()
Is Y already loaded in memory?

Returns Boolean

`load_random`()
Loads the random catalog with the realizations

`make_random`(*outfileran=''*, *ntimes=-1*)
Actually makes the random realizations :param str outfileran: output file (not needed) :param int ntimes: taken from class Pars unless otherwise indicated

`oob_data`(*frac=0.0*)
Creates oob data and separates it from the no-oob data for further tests :param float frac: Fraction of the data to be separated, taken from class Pars (default is 1/3)

`sample_dim`(*nsample*)
Samples from the list of attributes

Parameters **nsample** (*int*) – size of subsample

Returns dictionary with subsample attributes and their locations

`data.`**`create_random_realizations`**(*AT*, *F*, *N*, *keyatt*)
Create random realizations using error in magnitudes, saves a temporarily file on train data directory. Uses normal distribution

Parameters

- **AT** (*dict*) – dictionary with columns names and colum index
- **F** (*float*) – Training data
- **N** (*int*) – Number of realizations
- **keyatt** (*str*) – Attribute name to be predicted or classifed

Returns Returns an array with random realizations

`data.`**`make_AT`**(*cols*, *attributes*, *keyatt*)
Creates dictionary used on all routines

Note: Make sure all columns have different names, and error columns are the same as attribute columns with a 'e' in front of it, ex. 'mag_u' and 'emag_u'

Parameters

- **cols** (*str*) – str array with column names from file
- **attributes** (*str*) – attributes to be used from those columns
- **keyatt** (*str*) – Attribute to be predicted or classified

Returns dictionary, each key correspond to an attribute and itself a dictionary where 'ind' is the column index and 'eind' is the error column for the same attribute, ex., A={u:{'ind'=1, 'eind'=6}}

Return type dict

`data.`**`read_catalog`**(*filename*, *myrank=0*, *check='no'*)
Read the catalog, either for training or testing currently accepting ascii tables, numpy tables

Parameters

- **filename** (*str*) – Filename of the catalod
- **myrank** (*int*) – current processor id, for parallel reading (not implemented)
- **check** (*str*) – To check the code, only uses 50 lines of catalog

Returns The whole catalog

Return type float array

3.4.2 Utils

This is the `utils_mlz` class in some detail, this class contains some useful routines used by MLZ.

Module author: Matias Carrasco Kind

class `utils_mlz.`**`Stopwatch`**(*verb='yes'*)

Stopwatch and some time procedures

Parameters **verb** (*str*) – 'yes' or 'no' (verbose?)

`elapsed`(*only_sec=False*, *verbose=True*)

Prints and saves elapsed time

Parameters

- **only_sec** (*bool*) – set this to True for the elapsed time prints in seconds only
- **verbose** (*bool*) – Prints on screen

`restart`(*verb='Restart'*, *verbose=True*)

Set the counter to zero, keeps tracking of starting time

Parameters **verb** (*str*) – 'Start' (default) or 'Restart' (set the counter to zero and the starting

time to current time, keeps the initial starting in self.start0)

class `utils_mlz.`**`bias`**(*zs*, *zb*, *name*, *zmin*, *zmax*, *nbins*, *mode=1*, *d_z=<function <lambda> at 0x37536e0>*, *verb=True*)

Creates a instance to compute some metrics for the photo-z calculation for quick analysis

Parameters

- **zs** (*float*) – Spectrocopic redshift array
- **zb** (*float*) – Photometric redshift
- **name** (*str*) – name for identification
- **zmin** (*float*) – Minimum redshift for binning
- **zmax** (*float*) – Maximum redshift for binning
- **nbins** (*int*) – Number of bins used
- **mode** (*int*) – 0 (binning in spec-z) or 1 (binning in photo-z)
- **d_z** (*function*) – function to be applied on z_phot and z_spec, default (z_phot-z_spec)
- **verb** (*bool*) – verbose?

`utils_mlz.`**`compute_error`**(*z*, *pdf*, *zv*)

Computes the error in the PDF calculation using a reference values from PDF it computes the 68% percentile limit around this value

Parameters

- **z** (*float*) – redshift
- **pdf** (*float*) – photo-z PDF
- **zv** (*float*) – Reference value from PDF (can be mean, mode, median, etc.)

Returns error associated to reference value

Return type float

`utils_mlz.`**`compute_zConf`**(*z*, *pdf*, *zv*, *sigma*)

Computes the confidence level of the pdf with respect a reference value as the area between zv-sigma(1+zv) and zv+sigma(1+zv)

Parameters

- **z** (*float*) – redshift

- **pdf** (*float*) – photo-z PDF
- **zv** (*float*) – reference value
- **sigma** (*float*) – extent of confidence

Returns zConf

Return type float

class `utils_mlz.conf`(*zconf*, *zb*, *zmin*, *zmax*, *nbins*)

Computes confidence level (zConf) as a function of photometric redshift

Parameters

- **zconf** (*float*) – zConf array for galaxies
- **zb** (*float*) – Photometric redshifts
- **zmin** (*float*) – Minimum redshift for binning
- **zmax** (*float*) – Maximum redshift for binning
- **nbins** (*int*) – Number of bins used

`utils_mlz.get_area`(*z*, *pdf*, *z1*, *z2*)

Compute area under photo-z Pdf between z1 and z2, PDF must add to 1

Parameters

- **z** (*float*) – redshift
- **pdf** (*float*) – photo-z PDF
- **z1** (*float*) – Lower boundary
- **z2** (*float*) – Upper boundary

Returns area between z1 and z2

Return type float

`utils_mlz.get_limits`(*ntot*, *Nproc*, *rank*)

Get limits for farming an array to multiple processors

Parameters

- **ntot** (*int*) – Number of objects in array
- **Nproc** (*int*) – number of processor
- **rank** (*int*) – current processor id

Returns L1,L2 the limits of the array for given processor

Return type int, int

`utils_mlz.percentile`(*Nvals*, *percent*)

Find the percentile of a list of values. :param float Nvals: list of values :param float percent: a percentile value between 0 and 1. :return: percentile value :rtype: float

`utils_mlz.print_dtpars`(*DTpars*, *outfile*)

Prints the values from class Pars to a file

Parameters

- **DTpars** (*class*) – class Pars from input file
- **outfile** (*str*) – output filename

`utils_mlz.`**`read_dt_pars`**(*filein*, *verbose=True*, *myrank=0*)

Read parameters to be used by the program and convert them into integers/float if necessary, returning a class

Parameters

- **filein** (*str*) – name of inputs file, check format here *input-file*
- **verbose** (*bool*) – True or False
- **myrank** (*int*) – processor id for multi-core capabilities

`utils_mlz.`**`zconf_dist`**(*conf*, *nbins*)

Computes the distribution of Zconf for different bins between 0 and 1

Parameters

- **conf** (*float*) – zConf values
- **nbins** (*int*) – number of bins

Returns zConf dist, bins

Return type float,float

3.4.3 Plotting

This is the `plotting` class in some detail, this is used for plotting some results for the photometric redshift problem. Check *Running a test* to check a quick view on how to use this.

Module author: Matias Carrasco Kind

class `plotting.`**`Qplot`**(*inputs_file*)

Creates a qplot instance to produce a set of useful plot for quick analysis

Parameters **inputs_file** (*str*) – path to input file where all information and parameters are declared *input-file*

`plot_importance`(*result_id=0*, *Nzb=10*)

Plot ranking of importance of attributes used during the training/testing process

Note: The key *OobError* and *VarImportance* in *input-file* must be set to 'yes' to compute these quantities

Parameters

- **results_id** (*int*) – Result id number as the output on the results folder, default 0
- **Nzb** (*int*) – Number of redshift bins

`plot_map`(*nmap=0*, *colbar='yes'*, *min_m=-100*, *max_m=-100*)

Plot a map created during the training process,

Parameters

- **nmap** (*int*) – Number of created map, default is 0
- **min_m** (*float*) – Lower limit for coloring the cells, -100 uses min value
- **max_m** (*float*) – Upper limit for coloring the cells, -100 uses max value
- **colbar** (*str*) – Include a colorbar ('yes','no')

`plot_pdf_use`(*result_id=0*)
PLots the redshift distribution using PDFs and using one single estimator and a map of zphot vs zspec using also PDFs.

Note: The code utils/use_pdfs must be run first in order to create the needed files, it can be run in parallel

Parameters **result_id** (*int*) – result id (run number) as appears on the results , default = 0

`plot_results`(*result_1=0*, *zconf_1=0.0*, *result_2=0*, *zconf_2=0.0*)
Plots a summary of main results for photometric redshifts, it has user interactive plots.

Parameters

- **result_1** (*int*) – result id (run number) as appears on the results , default = 0 (uses mean of PDF for metrics)
- **zconf_1** (*float*) – confidence level cut for file 1
- **result_2** (*int*) – result id (run number) as appears on the results folder for a second optional file , default shows file 1 instead using the mode for the metrics
- **zconf_2** (*float*) – confidence level cut for file 2

`plot_tree`(*ntree=0*, *save_files='no'*, *fileroot='TPZ'*, *path=''*)
Plot a tree created during the training process, uses the Graphviz package (dot and neato)

Parameters

- **ntree** (*int*) – Number of created tree, default is 0
- **save_files** (*str*) – Saves the created files from Graphviz (the .png and the .dot files) 'yes'/'no'
- **fileroot** (*str*) – root name for saved files
- **path** (*str*) – path name for output files

3.5 Run MLZ

3.5.1 Input file

A brief explanation on how to run MLZ. The main code is included as a executable and can be called directly or its directory, its content can be viewed *here*

A self-explanatory view of the *input-file* is helpful to look at before running the code. This file can be used as a template for other files, the parameters can be checked in advance by setting `CheckOnly` to yes.

Note that the names of the variables are case insensitive but all of them need to be present.

3.5.2 Prepare data

Both the training file and test file must have the attributes (magnitudes, colors, etc.) and (optimally) their errors. If errors are not present assume a very small value is used. For now ascii files and numpy files (.npy) are valid. Spectroscopic redshifts must be included on the training file, if present on the test file they can be used for testing the performance of MLZ, although it is not required.

Add the full path relative to the working directory of these file to the input file and define a output folder for the results.

There are 3 very important variables on the input file to specify the columns and the attributes to use by separating them using comas. Make sure to indicate the spectroscopic redshift by its name in the `KeyAtt` variable. Also always indicate the name of the error columns by adding the letter `e` in front of the name of the attribute (see the *input-file* for an example)

In the `Att` variable, indicate the attributes to use to make a compute photo-z, you can add or remove attributes but make sure they are present on the columns names. Order is not important, but order in columns name are important

3.5.3 Some hidden parameters

In order to make the *input-file* not too busy, there are some hidden parameters in the `utils/utils_mlz.py` file that are not frequently used and can be manually modified, among the most important ones:

- oobfraction: fraction of data used for cross-validation, default is 1/3
- dotrain: The training of the trees/maps is carried out, default is yes, set it to no if want to use same trees or maps on a separate data set, it can save some time for large training data
- dotest: The test phase is carried out, default is yes, set it to no if only training is desired
- multiplefiles: Write a PDF file per core for large runs to avoid communications bottleneck, default is no
- writepdf: Write the PDF? default if yes, if not needed can be set to no

3.5.4 Run the code

Check the *Running a test* for a example use of the code on SDSS data including with the distribution.

To run the code, if using *mpi4py* from the main folder type:

```
$ mpirun -n <cores> ./runMLZ <input file>
```

Where <cores> is the number of processors desired to use and <input file> is the name of the *input-file*. If not using *mpi4py*, type:

```
$ ./runMLZ <input file>
```

Or if distribution is build or installed using pip, just type:

```
$ runMLZ <input file>
```

This will create two folder on the output directory, one named trees (or maps) where several files for trees or maps are stored for further analysis and the other folder named results where the main results are stored as well as the parameters used. The .mlz file contains 7 columns (zspec, zmode, zmean, zconf_mode, zcond_mean, error_mode, error_mean) which summarizes the results if no PDF is further needed. The PDF for all the galaxies are also stored in the same folder.

3.5.5 Machine learning approach

MLZ can be used through *TPZ* or *SOMz* and whichever is used is set on the *input-file* under the `PredictionMode` variable. Whether is a classification or a Regression problem this is set on the

`PredictionClass` variable. There are some variables common for both approaches and other exclusively used by one of them. For classification labels you can **must** use integers can can use the variable `MinZ` and `MaxZ` to enclose the range of values. OOB and cross-validation data are computed when the variable `OobError` is set to yes and a ranking of variable importance can be computed if the variable `VarImportance` is set to yes.

3.5.6 Preview of results

Some routines are provided to preview some results. See the *Running a test* and `plotting` for more information and some examples of figures that can be created

3.6 Running a test

3.6.1 Run on SDSS data

This distribution comes with a test folder where a example training set and a example testing set are located. This example correspond to a random subset of galaxies taken from the Main Galaxy Sample (MGS) from the SDSS. Each file has 5000 galaxies with spectroscopic redshift and magnitudes (model mag) and colors corrected by extinction in the 5 bands, u, g, r, i and z as well as their associated errors, making a total of 9 attributes. Make sure you look at *Run MLZ* for a general information on running MLZ

Note: This is a very small subsample of the whole catalog to illustrate the use of the MLZ and its capabilities. Also only few trees or maps are created for illustration, ideally hundreds of trees and maps are necessary

To run MLZ, type:

```
$ ./runMLZ test/SDSS_MGS.inputs
```

To run this example you must be located at the tpz/ folder, if using **mpi4py** type:

```
$ mpirun -n <cores>  ./runMLZ test/SDSS_MGS.inputs
```

Make sure <cores> matches your system. A view of the input file is *here*. The results are located in the folder mlz/test/results/ and the trees (or maps) are saved in tpz/test/trees/. There are some *other* parameters to control what phase to run or to manage the outputs.

3.6.2 Preview of results

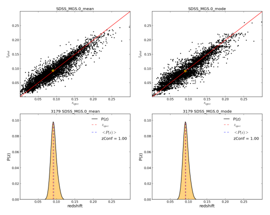
MLZ comes with some plotting routines, check `plotting` for some of them and their parameters. It includes an interactive routine to preview the results. Within the main folder type:

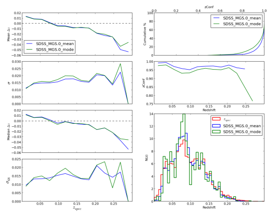
```
$ ./plot/plot_results test/SDSS_MGS.inputs 0 0
```

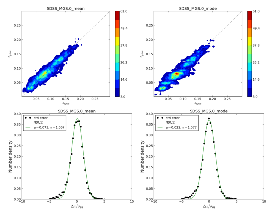
The first argument is the run number (every time TPZ increase this number by one) and the second argument is the confidence level zConf (see these *references*) for more information on this parameter and here for this routine `plotting.Qplot.plot_results`

Note: you can compare different runs (using different parameters for example) by adding two extra arguments with the number of the run and zConf for these results like **./plotting/plot_results.py test/SDSS_MGS.inputs 0 0 1 0** will show a comparison between the first and the second run with no zConf applied. If only 2 arguments are present after the input file, it shows a comparison for the mode and the mean for those results.

Three figures like the following are displayed for a summary of the results, with shape of PDFs, statisticsm etc







These figures have some user interaction as explained in the help window (shown below). For example by clicking different points in the zphot vs zspec figure is possible to visualize its PDF, or the colormap can be changed in figure 3, or change between zspec or zphot in the binning, etc...

```
All figures:
------------
* Q: close all figures
* q: close current figure

Figure 1:
---------
- Top Panel
* m, M: change color map
* +,- : change levels of countours
- Bottom Panel
* r/n : Toggle on/off Normal distribution
        with N(0,1)

Figure 2:
---------
* p/t : Toggle plots in photo bins and spectroscopic bins
* o   : Toggle on/off oob data when available

Figure 3:
---------
* Click on points to see its PDF
```

3.6.3 Plotting a tree or a map

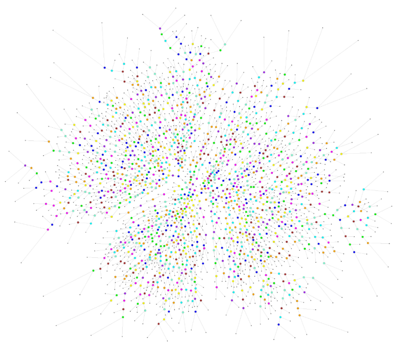
You can plot one of the created tree during the process in order to visualize how would it look like:

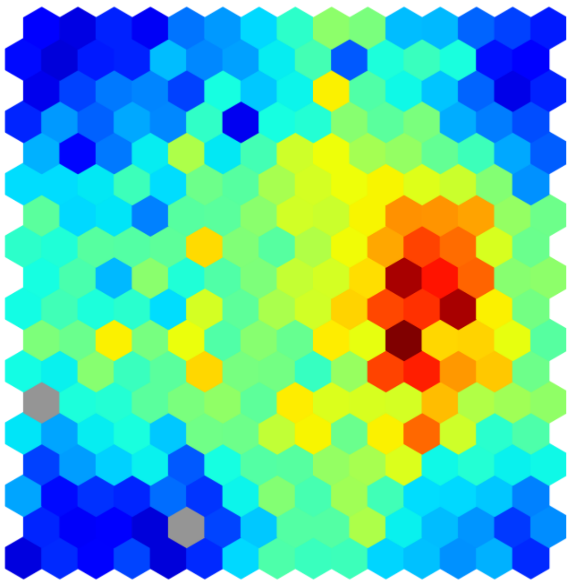
```
$ ./plot/plot_tree test/SDSS_MGS.inputs 0
```

Or if you used SOMz instead you can also plot a map using the following:

```
$ ./plot/plot_map test/SDSS_MGS.inputs 0
```

Check `plotting.Qplot.plot_tree` and `plotting.Qplot.plot_map` for information. The previous commands will generate figures like the following:





3.6.4 Some PDF example

Some examples on how to use the PDF to compute N(z) or a zphot vs zspec map there are some analysis routines for them, first we need to run some pre-analysis routine, if using **mpi4py** type:

```
$ mpirun -n <cores> ./utils/use_pdfs test/SDSS_MGS.inputs
```

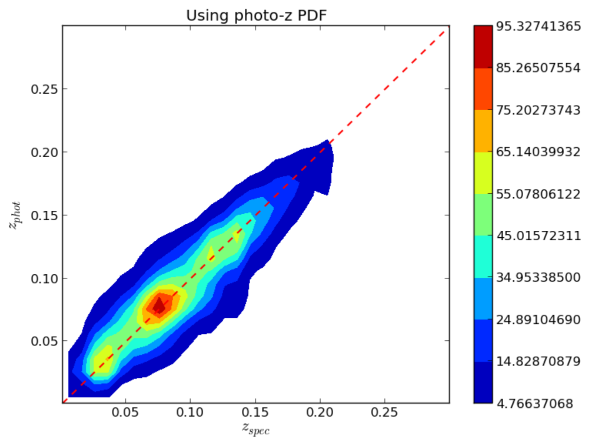
Making sure to enter the right number of cores, if using a serial version type:

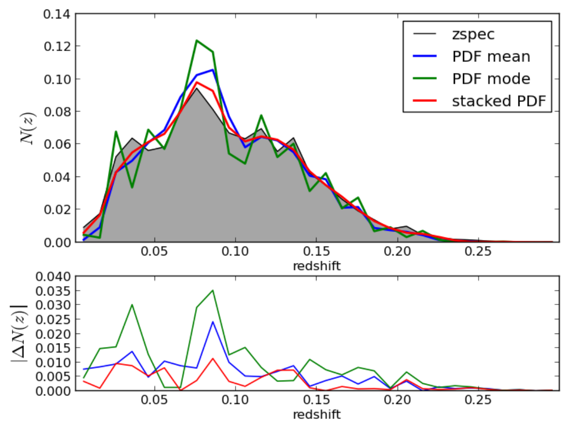
```
$ ./utils/use_pdfs test/SDSS_MGS.inputs
```

After this two extra files are created in the results folder with N(z) dist and a map, you can change the binning by modifying the use_pdfs file it self, by default is 30. To plot these you can check `plotting.Qplot.plot_pdf_use` and type:

```
$ ./plot/plot_pdf_use test/SDSS_MGS.inputs
```

And then you will see two figures like the following:



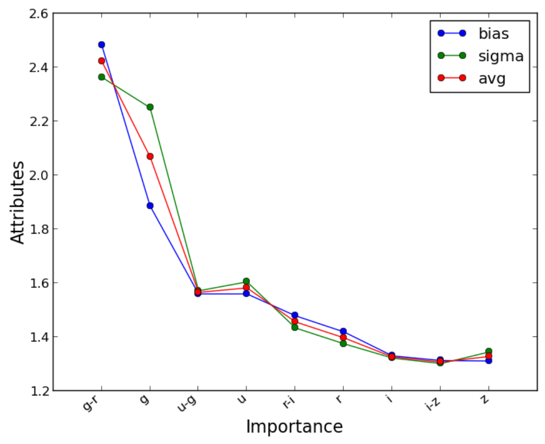


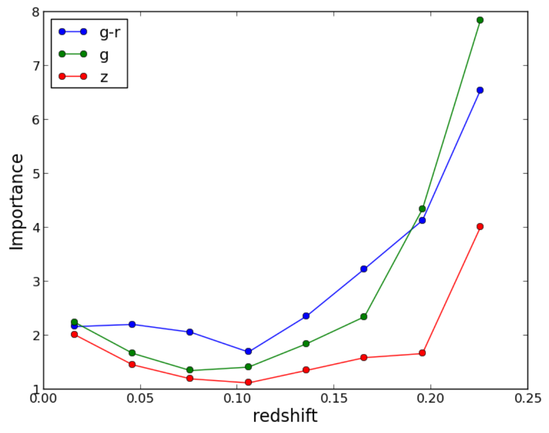
3.6.5 Ancillary information

If the extra information is set on the *input-file*, i.e., `OobError` and `VarImportance` are set to yes, then extra information can be plotted as well, note thar these variables are independent and setting only `OobError` to yes is always recommended as is a unbiased version of the performance on the same training set which serves a s a cross-validation and can be very useful. To plot the importance check first `plotting.Qplot.plot_importance` and type within the main folder:

```
$ ./plot/plot_importance test/SDSS_MGS.inputs
```

Which generated two plots like the following:





3.6.6 Extra notes

These figures and commands are only an example on how to run and visualize the data, these are not the optimal set of parameters for every data sets, look at the *references* for more information on what are the best parameters and suggestion to take advantage of MLZ, increasing the number of trees or the resolution for SOMz (`Ntop`) always help, `Natt` is also important, for TPZ one could start with the square root of the number of attributes and for SOM with 2/3 of the number of attributes. Email me at mcarras2 at illinois.edu for questions or comments

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX