Error analysis

For performing error analysis module error_analysis is used. To download module:

```
from citros_data_analysis import error_analysis as analysis
```

But first of all, let's have a quick look at data_access module, which is dedicate to query data.

Query data

To get access to a Citros database, create **CitrosDB** object:

```
from citros_data_analysis import data_access as da

citros = da.CitrosDB()
```

This way **CitrosDB** is created with defaults parameters. To specify connection parameters, pass correspondiong arguments:

```
citros = da.CitrosDB(host = 'hostName',
                     user = 'user',
                     password = 'myPassword',
                     database = 'myDatabase',
                     schema = 'mySchema',
                     batch = batchName,
                     port = '5432')
```

Data is always queried for exact topic. For example, to query all data for topic 'A':

```
df = citros.topic('A').data()
print(df)
```

▶ Show the result:

	sid	rid	time	topic	type	data.x.x_1	data.x.x_2	data.x.x_3	data.time	data.time	data.y
0	3	0	105036927	A	a	-0.080	-0.002	17.70	0.3	0.3	[2, 28, 45]
1	1	0	312751159	A	a	0.000	0.080	154.47	10.0	10.0	[15, 41, 43]
...	...	...	...	...	...	...	...	...	...	...	...

The result is a **DataFrame** of the **pandas** package.

Batch constists of two parts: json-data column, and all other columns. To query exact json-objects, pass list with their labels to **data()**. For example, if the json-data column looks like:

```
data
{'x': {'x_1' : 1, 'x_2' : 12, 'x_3' : 70}, 'y': [5.0, 3.4, 10], 'height' : 12}
{'x': {'x_1' : 5, 'x_2' : 10, 'x_3' : 73}, 'y': [5.5, 6.7, 50], 'height' : 11}
...
```

to query 'x_1', 'x_2', 'height' and values from the first position of 'y' json-array, the following code may be used:

```
df = citros.topic('A').data(['data.x.x_1', 'data.x.x_2', 'data.height', 'data.y[0]'])
```

Also, different constraints may be applied to query, see examples of data_access module.

Bin, interpolate and calculate statistics

CitrosData object

To perform data analysis the **CitrosData** object is used. Say, we would like to investigate the 'x' vector and its behaviour depending on the time. Let's query data:

```
df = citros.topic('A').data(['data.x', 'data.time'])
print(df)
```

▶ Show the output table:

	sid	rid	time	topic	type	data.x	data.time
0	3	0	105036927	A	a	{'x_1': -0.08, 'x_2': -0.002, 'x_3': 17.7}	0.3
1	1	0	312751159	A	a	{'x_1': 0.0, 'x_2': 0.08, 'x_3': 154.47}	10.0
...	...	...	...	...	...	...	...

CitrosData object has two main attributes - 'data' - the depending variables whose behavior we would like to study, and all other additional columns - 'addData'. It is possible to specify which column(s) to treat as data by `data_label`. You may also specify `units` of the data to make future plots more informative.

```
dataset = analysis.CitrosData(df,
                              data_label=['data.x'],
                              units = 'm')
```

When **CitrosData** object is created, it will turn all dicts or lists, if there were any in **df** rows, into separate columns and store them in 'data' attribute as a **DataFrame**:

```
print(dataset.data)
```

▶ Show the content of the 'data' attribute:

	data.x.x_1	data.x.x_2	data.x.x_3
0	-0.080	-0.002	17.70
1	0.000	0.080	154.47
...	...	...	...

All other data is stored in 'addData' attribute:

```
print(dataset.addData)
```

▶ Show the content of the 'addData' attribute:

	sid	rid	time	topic	type	data.time
0	3	0	105036927	A	a	0.3
1	1	0	312751159	A	a	10.0
...	...	...	...	...	...	...

It is possible to turn data of **CitrosData** object back to pandas.DataFrame. Method **to_pandas()** concatenate 'data' and 'addData' attributes and return the result table as a pandas.DataFrame:

```
F = dataset.to_pandas()
```

▶ The result of the concatenation:

	data.x.x_1	data.x.x_2	data.x.x_3	sid	rid	time	topic	type	data.time
0	-0.080	-0.002	17.70	3	0	105036927	A	a	0.3
1	0.000	0.080	154.47	1	0	312751159	A	a	10.0
...	...	...	...	...	...	...	...	...	...

Assigning indexes

To analyze data of multiple simulations it is necessary to establish a correspondence between the values of the data from these different simulations. One approach is to select an independent variable, define a scale that is common to all simulations and assign indexes on this scale. Then, the values of variables from different simulations will be connected by this independent variable.

There are two ways to perform index assignment: divide the independent variable into N ranges, assign an index to each interval, and calculate the averages of the data values for each simulation in each range, or scale the independent variable to the interval [0,1], define a new range of N points uniformly distributed from 0 to 1, and interpolate data points over this new interval. The first approach corresponds to the **bin_data()** method, while the second is implemented by the **scale_data()** method.

Bin data

Let's choose one of the parameter, say 'data.time', divide it into `n_bins` intervals and assign index to each of the interval. Then let's group values of the 'x' vector from the previous example according to this binning and calculate the mean values of 'x' for the each group. This procedure may be done by function **bin_data()**. To see the histogram and control number of counts falling in each bin, pass `show_fig` = True:

```
db_bin = dataset.bin_data(n_bins = 50,
                          param_label = 'data.time',
                          show_fig = True)
```

▶ Show the distribution:

The result is a **CitrosData** object too, whose `data` and `addData` attributes have two levels of indexes - the new obtained after binning indexes and 'sid'. Mean values of the vector 'x' are stored in 'data' attribute and values of the bin centers are stored in 'addData' attribute.

```
print(db_bin.data)
```

▶ Show the content of the 'data' attribute:

		data.x.x_1	**data.x.x_2**	**data.x.x_3**
data.time_id	sid			
0	1	0.00000	0.08000	154.470000
	2	-0.04460	0.06540	87.728000
	3	-0.07900	0.00975	68.055000
1	1	0.01600	0.07800	74.453333
	2	-0.01600	0.07800	65.020000
...	...	...	...	...

```
print(db_bin.addData)
```

▶ Show the content of the 'addData' attribute:

		data.time
data.time_id	sid	
0	1	8.458
	2	8.458
	3	8.458
1	1	24.774
	2	24.774
...	...	...

Scale data

Another approach besides from binning is to scale parameter to [0,1] interval and interpolate data on this new interval with equally spaced points. Data of different 'sid' values processed independently. The function to perform this is **scale_data**. It's syntax is pretty similar to **bin_data()**:

```
db_sc = dataset.scale_data(n_points = 50,
                                param_label = 'data.time',
                                show_fig = False)
```

Structure of the output is similar too:

```
print(db_sc.data)
```

▶ Show the content of the 'data' attribute:

		data.x.x_1	data.x.x_2	data.x.x_3
data.time_id	sid			
0	1	0.000000	0.080000	154.470000
	2	-0.057000	0.056000	108.950000
	3	-0.080000	-0.002000	17.700000
1	1	0.025494	0.075573	6.569425
	2	-0.028482	0.074719	167.725144
...	...	...	...	...

```
print(db_sc.addData)
```

▶ Show the content of the 'addData' attribute:

		data.time
data.time_id	sid	
0	1	0.000000
	2	0.000000
	3	0.000000
1	1	0.020408
	2	0.020408
...	...	...

As previously in the case of **bin_data()** method, to controll if the number of point should be increased, pass `show_fig` = True and the result of interpolation for each of the vector's component will be shown. Plots are shown for the first five 'sid' values.

▶ Show example for example the 'x_1':

Statistics

Get statistics

Now, when we bin or scale data over one of the independent parameter and set new indices according to these procedures, we are able to study statistics for each of these indices. **get_statistics()** method is dedicated to do it:

```
stat = db_sc.get_statistics(return_format='citrosStat')
```

It returns **CitrosStat** object. Its attributes store independent parameter (stat.x), mean values (stat.mean), covariant matrix (stat.covar_matrix) and standard deviation (stat.sigma, the square root of its diagonal elements) calculated over sids for each index. Each attribute is a **pandas.DataFrame**, except for covariant matrix, which is a **pandas.Series**:

```
print(stat.mean)
```

	data.x.x_1	data.x.x_2	data.x.x_3
data.time_id			
0	-4.56666667e-02	4.46666667e-02	9.37066667e+01
...	...	...	...

```
print(stat.x)
```

	data.time
data.time_id	
0	0.000000
1	0.020408
...	...

```
print(stat.sigma)
```

	data.x.x_1	data.x.x_2	data.x.x_3
data.time_id			
0	4.11865674e-02	4.21584313e-02	6.96475242e+01
...	...	...	...

Covariant matrix containes N x N numpy.ndarray, with N being the dimension of the data:

```
>>> print(stat.covar_matrix.iloc[0])

array([[1.69633333e-03, 1.54366667e-03, 2.60583167e+00],
       [1.54366667e-03, 1.77733333e-03, 2.93335333e+00],
       [2.60583167e+00, 2.93335333e+00, 4.85077763e+03]])
```

get_statistics() method may return statistics as a single **DataFrame**:

```
stat = db_sc.get_statistics(return_format='pandas')
print(stat)
```

	data.time	mean	covar_matrix	sigma
data.time_id				
0	0.000000	array([...	array([[...	array([...
1	0.020408	array([...	array([[...	array([...
...	...	...	...	...

That way, the type of 'mean', 'covar_matrix' and 'sigma' values in each row are numpy.ndarray:

```
>>> print(stat['covar_matrix'].iloc[0])

array([[1.69633333e-03, 1.54366667e-03, 2.60583167e+00],
       [1.54366667e-03, 1.77733333e-03, 2.93335333e+00],
       [2.60583167e+00, 2.93335333e+00, 4.85077763e+03]])
```

```
>>> print(stat['mean'].iloc[0])

array[-4.56666667e-02  4.46666667e-02  9.37066667e+01]
```

```
>>> print(stat['sigma'].iloc[0])

array([4.11865674e-02, 4.21584313e-02, 6.96475242e+01])
```

Plot statistics

To visualize statistics **show_statistics()** function is used. It plots values from `data` attribute vs. independent parameter for each of the sid, the mean value over all sids and 3 σ interval. If 'data' has several components, like in the example above ('x_1', 'x_1', 'x_3'), it makes separate plots for each of the component:

```
db_sc.show_statistics()
```

▶ Show the statistics plot:

3 σ standard deviation interval is represented by red lines. To change the number of standard deviations, for example to plot 2 σ boundary, pass `n_std` = 2. Set parameter `std_area` = True to fill the area the boundary and `std_lines` = False to remove the border lines:

```
db_sc.show_statistics(std_area = True, std_lines = False, n_std = 2)
```

To study in details the features at the exact 'data.time' value see about **show_correlation()** method.

Correlation

Function **show_correlation()** plots correlation between two variables for the exact index `slice_id`. Applying it to the previous example for the time_id = 0:

```
db_sc.show_correlation(x_col = 'data.x.x_1',
                       y_col = 'data.x.x_2',
                       slice_id = 0,
                       n_std = [1,2,3])
```

The value of the independent parameter, that corresponds to `slice_id` = 0 is 0.0. This information is printed in the output:

```
slice_id = 0,
slice_val = 0.0
```

▶ Show the correalation plot:

Pass to `x_col` and `y_col` either label or index of the columns to plot along x and y axis respectively. Instead of `slice_id` index the value `slice_val` may be specified. That way, the nearest `slice_id` index will be found and the corresponding to it exact value of `slice_val` will be printed. `n_std` states the radious or radii of the confidence ellipses in sigmas. If `bounding_error` = True, then the bounding error circle is added to plot.

The following code plots first column ('data.x.x_1') vs. second ('data.x.x_2') for the id, nearest to 'data.time' = 0.2 and plots bounding error circle.

```
db_sc.show_correlation(x_col = 0,
                       y_col = 1,
                       slice_val = 0.2,
```

```
                        n_std = [1,2,3],
                        bounding_error = True)
```

The nearest to `slice_val` = 0.2 `slice_id` turned out to be 10 and the exact value, that corresponds to `slice_id` = 10 is 0.204:

```
slice_id = 10,
slice_val = 0.204
```

▶ Show the correalation plot with bounding error:

To plot correlation between variables from different **CitrosData** objects, pass the object by `db2` parameter. This way, `x_col` is supposed to be the column of the first **CitrosData** objects, while `y_col` - the column of the `db2`. If `slice_val` is passed, then the `slice_id` is searched for each **CitrosData** objects.

Let's query for topic 'B', scale it over 20 points and plot correlation between 'data.x.x_2' of `db_sc` vs. 'data.x.x_1' of `db_sc_B` near the point 'data.time' = 0.7:

```
#download columns 'data.x' and 'data.time' for topic 'B'
df_B = citros.topic('B').data(['data.x', 'data.time'])

#construct CitrosData object with 3 data-columns from 'data.x'
dataset_B = analysis.CitrosData(df, data_label=['data.x'], units = 'm')

#scale data
db_sc_B = dataset.scale_data(n_points = 20,
                             param_label = 'data.time',
                             show_fig = False)

db_sc.show_correlation(db2 = db_sc_B,
                       x_col = 'data.x.x_2',
                       y_col = 'data.x.x_1',
                       slice_val = 0.7,
                       n_std = [1,2,3])
```

The output is:

```
slice_id = 34,
slice_val = 0.694,
slice_id_2 = 13,
slice_val_2 = 0.684
```

▶ Show the plot:

Regression

Different input parameters may vary the output, and to predict how the parameter affects the result, regressions are used. By now, three methods of regression are supported: polinomial regression ('poly'), simple neural net regression ('neural_net') and gaussian mixture model ('gmm'). To apply regression analysis, we need to construct **CitrosDataArray** object, that stores **CitrosData** objects with different input parameters.

```
db_array = analysis.CitrosDataArray()
```

Let's say for the topic 'A' we have data for four different values of the parameter 't', that is written in json-data column 'data.t'. First, let's get all possible 'data.t' values for topic 'A' (see examples of data_access module):

```
>>> list_t = citros.topic('A').get_unique_values('data.t')
>>> print(list_t)

[-1.5, 0, 2.5, 4]
```

Now let's query data for each of these parameter values, set it as parameter, scale data over 'data.time' and put to **CitrosDataArray**:

```
for t in list_t:
    #query data
    df = citros.topic('A').\
                set_filter({'data.t': [t]}).\
                data(['data.x.x_1', 'data.time', 'data.p', 'data.t'])

    #create CitrosData object and set 'data.t' as a parameter.
    dataset = analysis.CitrosData(df,
                                  data_label=['data.x.x_1'],
                                  units = 'm',
                                  parameter_label = ['data.t'])

    #scale over 'data.time'
    db_sc = dataset.scale_data(n_points = 100,
                               param_label = 'data.time',
                               show_fig = False)

    #store in CitrosDataArray by add_db() method
    db_array.add_db(db_sc)
```

To set value from 'data.t' column as a parameter, pass the column label to `parameter_label` argument of **CitrosData** object.

If a column with parameters is not presented, it is possible to put `dict` with parameters by `parameters` argument:

```
dataset = analysis.CitrosData(df,
                              data_label=['data.x.x_1'],
                              units = 'm',
                              parameters = {'data.t': t})
```

or set parameters manualy by `set_parameter(` method. It accepts either setting parameter one by one by `key` and `value`:

```
db_sc.set_parameter(key = 'data.t', value = 0)
```

or by passing dictionary by `item`:

```
db_sc.set_parameter(item={'data.t': 0})
```

The last method allows you to pass multiple parameters, for example:

```
db_sc.set_parameter(item={'param_1':0, 'param_2':1, 'param_3':2})
```

Let's take a look at the scaled to [0,1] interval data:

```
import matplotlib.pyplot as plt

for db in db_array.dbs:
    plt.plot(db.addData['data.time'], db.data['data.x.x_1'],
             '.', label = 'data.t = ' + str(db.parameters['data.t']))
plt.xlabel('data.time')
plt.ylabel('data.x.x_1')
plt.legend(bbox_to_anchor=(1.0, 1.0))
```

It is a sine function with some noise added, biased by 'data.t' value.

▶ Show the figure:

Let's find the solution for the case 'data.t' = 1 by **get_prediction()** method. Parameter label (in our case 'data.t') and value (1), for which the prediction is desired, should be passed as `dict` by `parameters` argument. Method of regression calculation ('poly' for polinomial regression, simple 'neural_net' for neural net regression and 'gmm' for gaussian mixture model) should be stated by `method` argument.

Polynomial regression

The first method is a polynomial regression, with the highest polynomial order defined by `n_poly`:

```
result = db_array.get_prediction(parameters = {'data.t': 1},
                                 method = 'poly',
                                 n_poly = 2,
                                 show_fig = True)
```

▶ Show the figure:

The result is a **DataFrame**.

▶ Show the result:

	data.time	data.x.x_1
0	0.000000	1.155301
1	0.010101	1.145971
2	0.020202	1.232255
...	...	...

Neural net

The second method is 'neural_net', which is based on the `sklearn.neural_network.MLPRegressor` class. Its the most important arguments are `activation`, that defines activation function for the hidden layers ('relu', 'identity', 'logistic' or 'tanh'), `max_iter` - maximum number of iteration, `solver` - solver for weight optimization ('lbfgs', 'sgd' or 'adam'), `hidden_layer_sizes` - the number of neurons in the ith hidden layer, see `sklearn.neural_network.MLPRegressor` for the details of the other possible arguments.

```
result = db_array.get_prediction(parameters = {'data.t':1},
                                 method = 'neural_net',
                                 activation='tanh', max_iter = 200, solver='lbfgs',
                                 hidden_layer_sizes = (8,), random_state = 9,
                                 show_fig = True)
```

▶ Show the figure:

Gaussian mixture model

The last method is a gaussian mixture model:

```
result = db_array.get_prediction(parameters = {'data.t':1},
                                 method = 'gmm',
                                 show_fig = True)
```

▶ Show the result:

Regression comparison

To compare the results of these methods, list their names as `method`:

```
result_list = db_array.get_prediction(parameters = {'data.t':1},
                                      method = ['neural_net', 'poly', 'gmm'],
                                      n_poly = 2,
```

```
                                        activation='tanh', solver='lbfgs', hidden_layer_sizes = (8,),
random_state = 9,
                                        show_fig = True)
```

▶ The resulting plot:

That way, the returning result is a list of the **DataFrames**.