

demo

June 21, 2026

1 User guide for the ddsimca package

This guide is written for the conventional PCA/SVD-based DD-SIMCA method. For the DD-SIMCA method developed for the analysis of 3-way data, see [demo_3way.ipynb](#).

The examples below are mainly based on the [DD-SIMCA tutorial paper](#), and most of the code reproduces outcomes and figures shown in the paper. It is therefore highly recommended to download and read the paper first (it is freely available) and then come back to this document.

Before you start, make sure that you have installed the `ddsimca` package. If not, just uncomment and run the following code:

```
[1]: #!pip install ddsimca
```

It will automatically install all necessary packages, including `prcv`, which implements Procrustes cross-validation, to be used later.

1.1 Training a DD-SIMCA model and detection of outliers

[Download](#) the zip archive with the Oregano dataset used in the tutorial. It consists of several CSV files; just unzip them all to the same folder where you have this document.

The following code loads the training set from a CSV file and shows the first five rows and five columns of the set:

```
[3]: import pandas as pd
import matplotlib.pyplot as plt
data_train = pd.read_csv("Target_Train.csv", index_col=0)

print(data_train.iloc[:5, :5])
```

	Class	8998.367	8990.654	8982.939	8975.226
Name					
Bio01	Oregano	-1.036600	-1.036965	-1.034487	-1.032063
Bio02	Oregano	-1.040406	-1.035694	-1.035552	-1.035854
Bio03	Oregano	-1.042949	-1.041892	-1.039863	-1.041063
Bio04	Oregano	-1.052614	-1.050210	-1.050617	-1.052789
Bio05	Oregano	-1.034859	-1.032732	-1.033896	-1.032058

As you can see, the data rows have object labels (hence we used `index_col=0` when loading the data). The first column of the dataset contains the class labels. For the training set, this column

should contain only the target class label (in our case "Oregano"); if there are more labels, or no labels at all, you will get an error when trying to create a model. The rest of the data frame consists of NIR spectra, already preprocessed.

Now we are ready to train the DD-SIMCA model and look at the summary info:

```
[4]: from ddsimca import ddsimca
m = ddsimca(data_train, ncomp = 10)
m.summary()
```

DDSIMCA model:

- target class: Oregano
- number of components (total): 10
- number of components (optimal): 10
- number of training samples: 52
- number of variables: 649
- preprocessing: mean centering

Parameters for classic estimators:

Comp	Nh	Nq	eigenvals
1	1	7	1.134
2	1	4	0.503
3	1	4	0.085
4	1	3	0.073
5	2	3	0.024
6	2	3	0.013
7	2	6	0.011
8	2	6	0.006
9	3	5	0.004
10	3	12	0.003

As you can see, the value 10 is the total number of components to use in the model. The optimal number can be discovered and set later; by default it is the same as the total number.

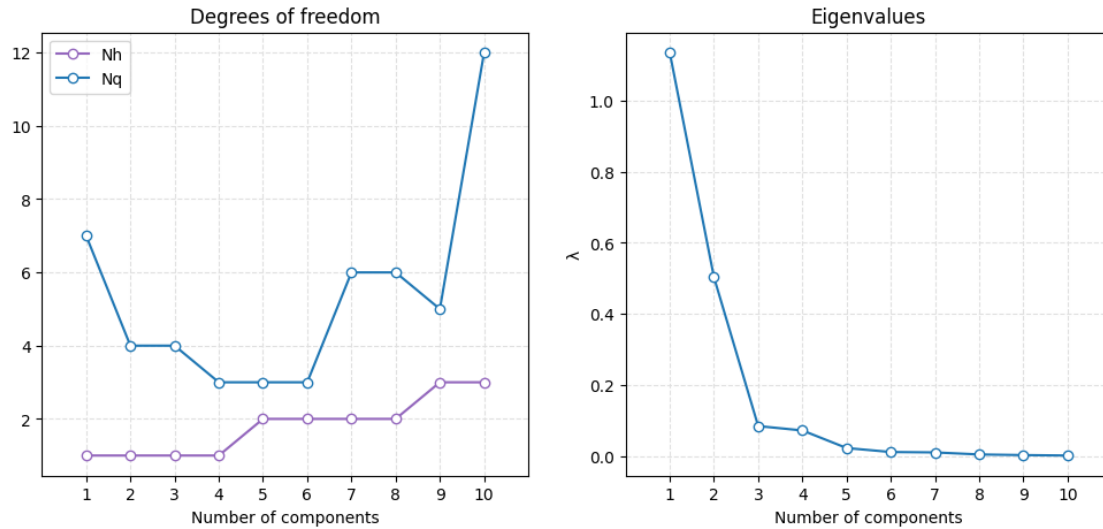
Also, by default the data values are mean-centered but not standardized. This can be changed by providing extra arguments — see the help for `ddsimca()` for more details.

You can visualize the number of degrees of freedom and the eigenvalues vs. the number of PCs using plots (eigenvalues can also be shown log-transformed — see the help for the method):

```
[5]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
m.plotDoF(ax1, dof = "Nh")
m.plotDoF(ax1, dof = "Nq")

ax2 = plt.subplot(1, 2, 2)
m.plotEigenvals(ax2)
```



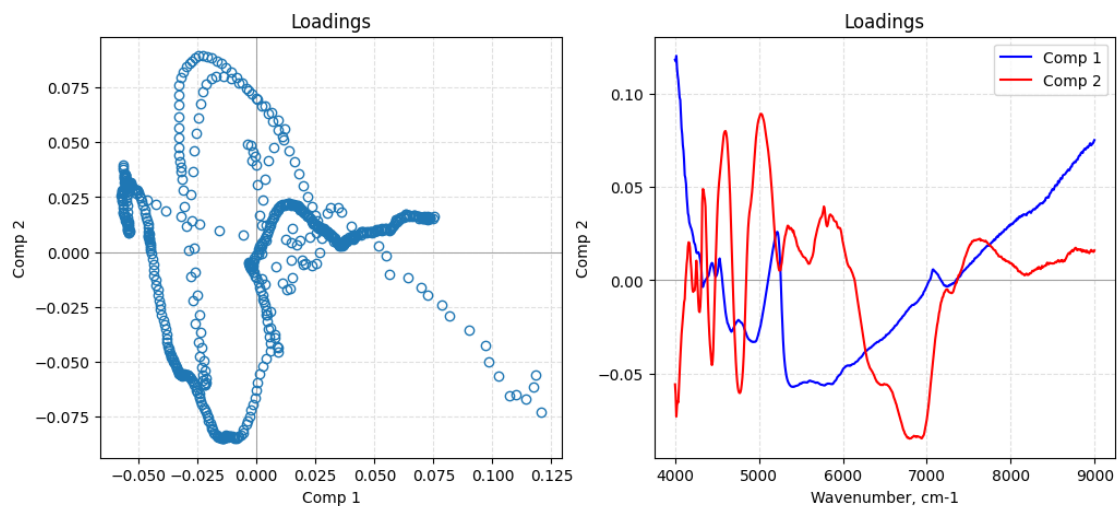
And show plots with the PCA loadings:

```
[6]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
m.plotLoadings(ax1, comp = (1, 2), type = "p")

ax2 = plt.subplot(1, 2, 2)
m.plotLoadings(ax2, comp = (1,), type = "l", color = "blue")
m.plotLoadings(ax2, comp = (2,), type = "l", color = "red")
ax2.legend()
ax2.set_xlabel("Wavenumber, cm-1")
```

```
[6]: Text(0.5, 0, 'Wavenumber, cm-1')
```



Here, the parameter `type` tells how to show the loadings: "p" stands for points (scatter plot), and "l" stands for lines. The parameter `comp` should be a tuple with two values for a scatter plot, and one value for a line plot, as shown above.

The model object does not have any results; it only contains values and statistics needed for applying this model to any dataset (e.g. loadings, vectors for centering and scaling, parameters of the distance distributions, etc.). To get the results, you need to apply this model to a dataset. Here is how to do it for the training set:

```
[7]: r_train_c = m.predict(data_train)
      r_train_c.summary()
```

DDSIMCA results:

```
- number of components (total): 10
- number of components (selected): 10
- limit type: classic
- alpha: 0.050
- gamma: 0.010

- class labels: provided
- number of objects: 52
- number of members: 52
- number of strangers: 0
```

PCs	eCrit	oCrit	in	out	TP	FN	sens
1	15.507	30.220	48	4	48	4	0.923
2	11.070	24.263	48	4	48	4	0.923
3	11.070	24.263	49	3	49	3	0.942
4	9.488	22.079	48	4	48	4	0.923
5	11.070	24.263	48	4	48	4	0.923
6	11.070	24.263	48	4	48	4	0.923
7	15.507	30.220	48	4	48	4	0.923
8	15.507	30.220	51	1	51	1	0.981
9	15.507	30.220	49	3	49	3	0.942
10	24.996	42.425	50	2	50	2	0.962

As you can see, most of the parameters — like the significance levels for extremes (`alpha`) and outliers (`gamma`), and the type of distance-limit estimator (`lim_type`) — are set to default values (0.05, 0.01, and "classic" respectively). If you want to change any of them, simply provide the proper values as arguments of the method `predict()`.

For example, let's create the result object using robust estimators:

```
[8]: r_train_r = m.predict(data_train, lim_type = "robust")
```

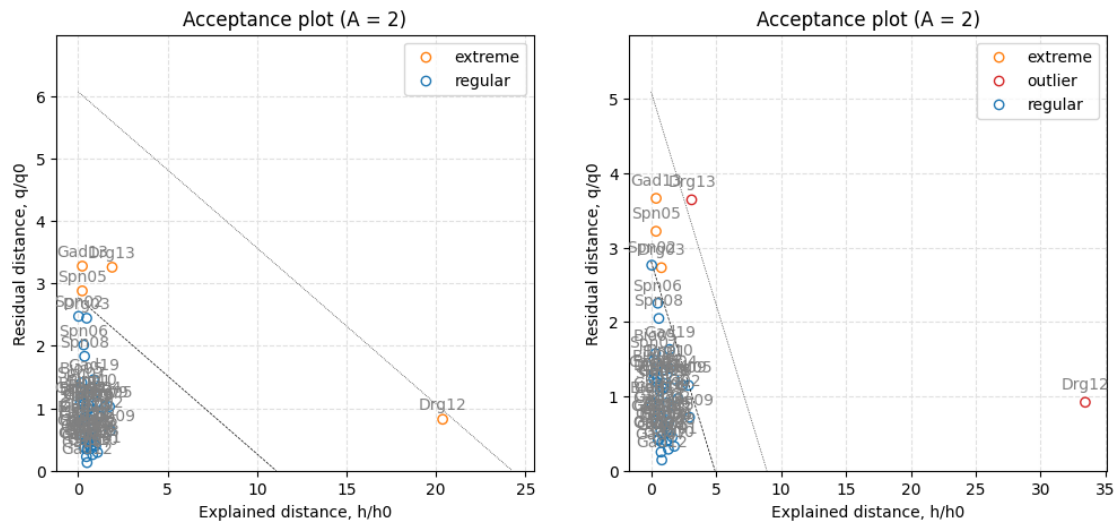
Now let's check the acceptance plot for both result objects (obtained using classic and robust estimators) in order to find any outliers. The figure below shows plots similar to (A) and (B) from

Figure 2 of the paper.

```
[9]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_train_c.plotAcceptance(ax1, ncomp = 2, show_labels = True)

ax2 = plt.subplot(1, 2, 2)
r_train_r.plotAcceptance(ax2, ncomp = 2, show_labels = True)
```

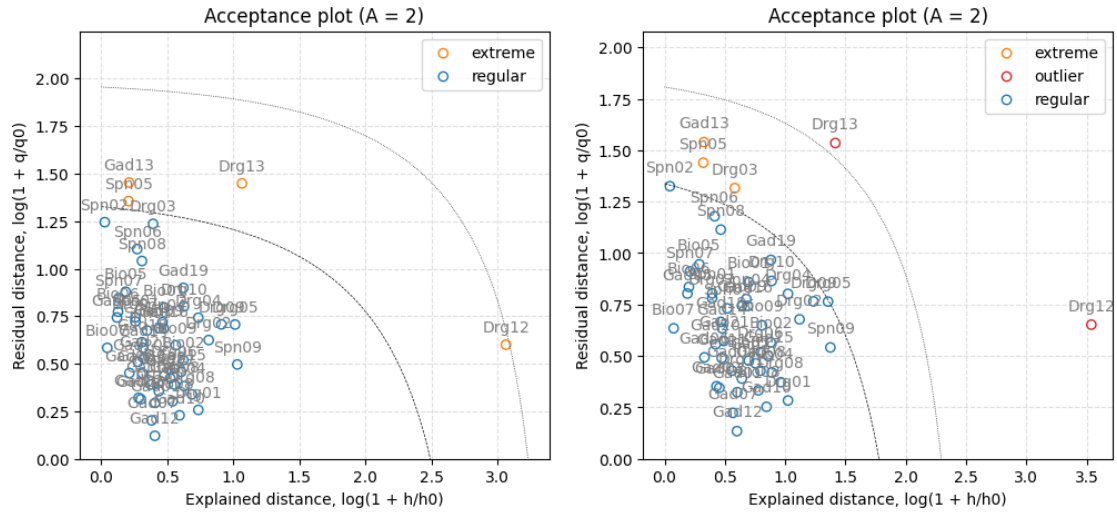


We can also show these two plots using log-transformed coordinates, like in Figure 3 of the paper:

```
[10]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_train_c.plotAcceptance(ax1, ncomp = 2, do_log = True, show_labels = True)

ax2 = plt.subplot(1, 2, 2)
r_train_r.plotAcceptance(ax2, ncomp = 2, do_log = True, show_labels = True)
```



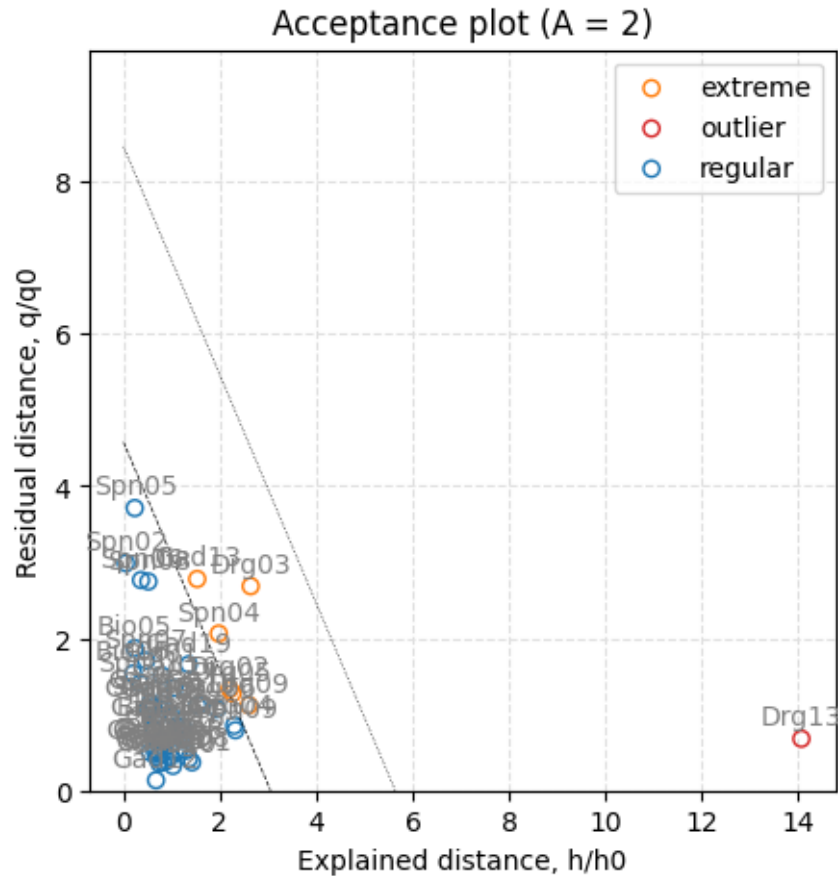
Apparently, as described in the paper, we need to remove the sample **Drg12** first, and then **Drg13**. Let's do this step by step and reproduce plots (C) and (D) of Figure 2. Note that every time we remove an outlier we need to re-train the model.

First, remove **Drg12** and reproduce plot (C):

```
[11]: data_train_new = data_train.drop("Drg12")

m_new = ddsimca(data_train_new, 10)
r_train_new = m_new.predict(data_train_new, lim_type = "robust")

plt.figure(figsize = (5, 5))
ax = plt.subplot(1, 1, 1)
r_train_new.plotAcceptance(ax, ncomp = 2, show_labels = True)
```

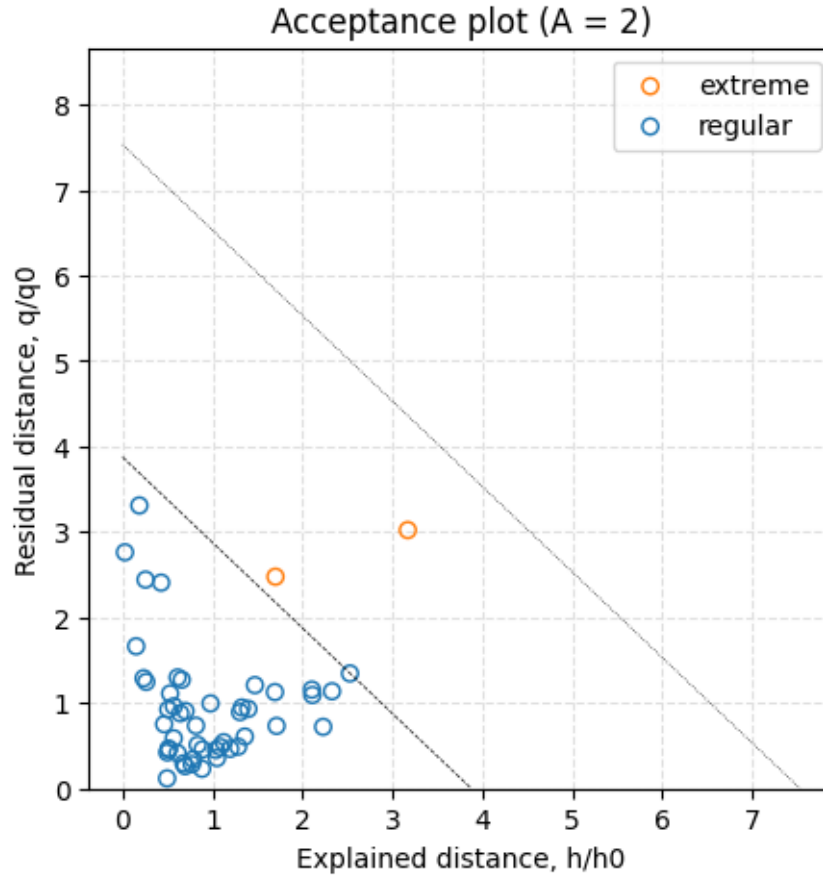


Now let's remove **Drg13** and reproduce plot (D) of Figure 2. Note that in this case we go back to classic estimators — from the paper we know that there are no more outliers in the data. Otherwise, it would be a good idea to use robust estimators again, and to check this plot for different numbers of components.

```
[12]: data_train_final = data_train_new.drop("Drg13")

m_final = ddsimca(data_train_final, 10)
r_train_final = m_final.predict(data_train_final)

plt.figure(figsize = (5, 5))
ax = plt.subplot(1, 1, 1)
r_train_final.plotAcceptance(ax, 2)
```



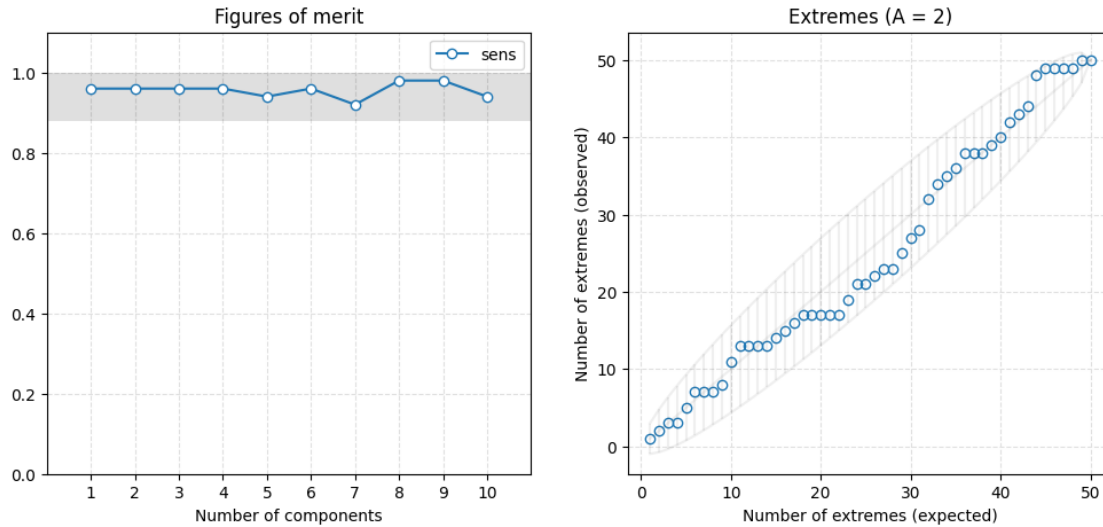
Finally, here are the sensitivity and extremes plots made for the training set, similar to what is shown in Figure 4 in the paper. There is one difference: in the paper we used $A = 20$ components in the model, while here we use 10 (to make the summary shorter). Therefore the sensitivity plot below is shown for the first 10 components only.

Here we use a versatile method, `plotFoM()`, which can show a plot for any of the five figures of merit: sensitivity ("`sens`"), specificity ("`spec`"), selectivity ("`sel`"), accuracy ("`acc`"), and efficiency ("`eff`").

```
[13]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_train_final.plotFoM(ax1, fom = "sens", show_ci = True)

ax2 = plt.subplot(1, 2, 2)
r_train_final.plotExtremes(ax2, ncomp = 2)
```

When the FoM plot is made for sensitivity, you can add a 95% confidence interval — computed from the expected sensitivity ($1 - \alpha$) and the number of objects in the dataset — shown as a semi-transparent rectangle on the plot above. This matches what is shown in the paper and what is implemented in the web application.

1.2 Validation and optimization

The best validation strategy is to use an independent validation set. However, if you want to keep it for the final testing of your model (or for fine-tuning) you can employ [Procrustes cross-validation](#), PCV. PCV is a procedure for generating a validation set based on the training set and cross-validation resampling. The code below does this based on the PCA version of the method, implemented in the `prcv` package:

```
[14]: from prcv.methods import pcvpca

# get matrix with predictors from the training set
X_train = data_train_final.iloc[:, 1:].values

# generate a matrix with the PV-set using 20 PCs, mean centering, and
↳ cross-validation
# with systematic splits (venetian blinds) into 4 segments
X_pv = pcvpca(X_train, ncomp = 20, center = True, scale = False, cv = {"type":
↳ "ven", "nseg": 4})

# create a data frame from the generated data
data_pv = pd.DataFrame(X_pv, index = data_train_final.index)
data_pv.insert(0, "Class", data_train_final.Class)
data_pv.columns = data_train_final.columns

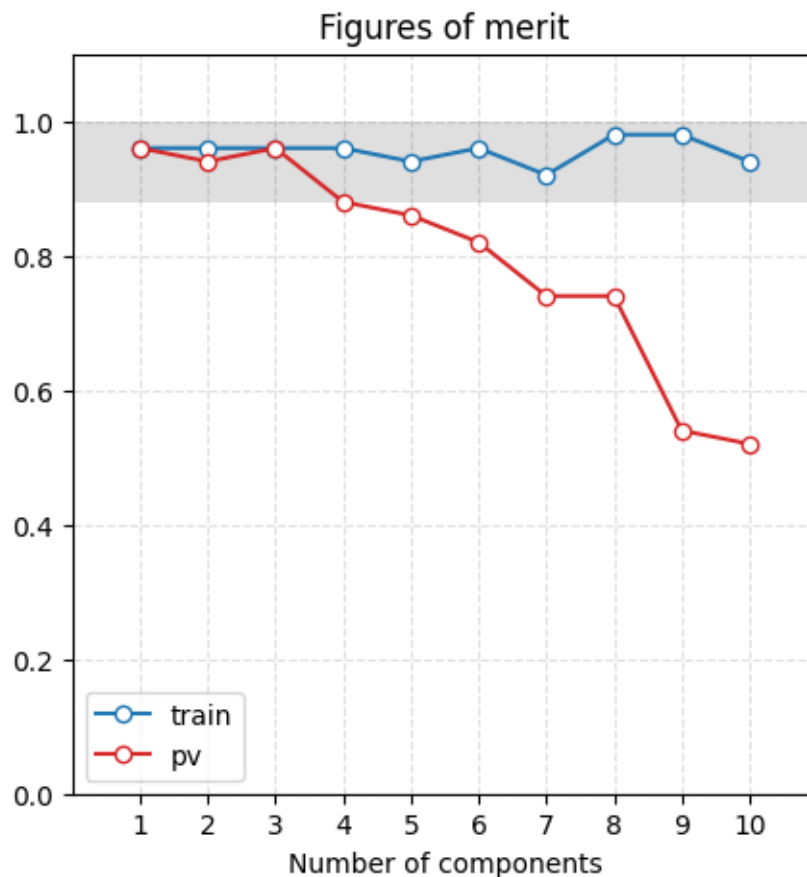
data_pv.iloc[:5, :5]
```

```
[14]:      Class  8998.367  8990.654  8982.939  8975.226
      Name
      Bio01  Oregano -0.973861 -0.974887 -0.975618 -0.977257
      Bio02  Oregano -1.044314 -1.041772 -1.041009 -1.040905
      Bio03  Oregano -1.030152 -1.030432 -1.026882 -1.028715
      Bio04  Oregano -1.050680 -1.046101 -1.046788 -1.048382
      Bio05  Oregano -0.963546 -0.962786 -0.966556 -0.967751
```

Now let's apply the model to the PV-set and show a combined sensitivity plot for the training set and for the PV-set, hence reproducing plot (A) from Figure 4:

```
[15]: r_pv = m_final.predict(data_pv)

plt.figure(figsize = (5, 5))
ax = plt.subplot(1, 1, 1)
r_train_final.plotFoM(ax, fom = "sens", label = "train", show_ci = True)
r_pv.plotFoM(ax, fom = "sens", label = "pv", color = "tab:red")
```



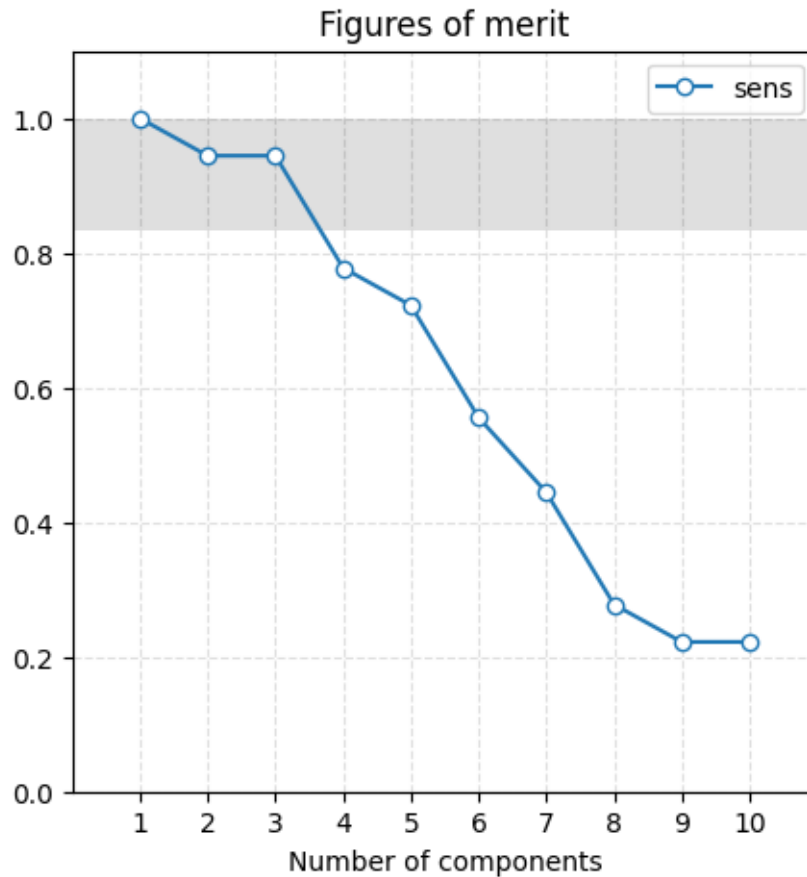
It looks like 3 components is optimal in this case. Let's now load the test set, which consists only of target class members, and apply the model to this set as well. Then we will show the sensitivity

plot for the test set, reproducing plot (B) of Figure 4.

```
[16]: data_test_target = pd.read_csv("Target_Test.csv", index_col=0)

r_test_target = m_final.predict(data_test_target)

plt.figure(figsize = (5, 5))
ax = plt.subplot(1, 1, 1)
r_test_target.plotFoM(ax, fom = "sens", show_ci = True)
```



Finally, you can also specify the optimal number of components for a model and for any result object. In this case, every time you make an acceptance plot (or perform any other action that depends on the number of components in the model), this value will be used as the default:

```
[17]: m_final.select_ncomp(3)
r_test_target.select_ncomp(3)
```

1.3 Predictions

Predictions can be made using data frames with or without reference class labels. In the first case, the result object will contain all necessary figures of merit (sensitivity for members, specificity and selectivity for non-members, and accuracy and efficiency if the dataset contains both members and strangers). If reference classes are not provided, the model will simply make predictions (accepted / rejected).

Let's load the dataset with reference classes (only non-target objects) and then remove the column with class names, thus creating a dataset without reference classes:

```
[18]: data_test_nontarget = pd.read_csv("NonTarget_Non_Or.csv", index_col = 0)
      data_new_nontarget = data_test_nontarget.iloc[:, 1:]
```

Let's check what is inside the datasets:

```
[19]: data_test_nontarget.iloc[:5, :5]
```

```
[19]:      Class  8998.367  8990.654  8982.939  8975.226
      Name
Bas01  Non_Or -0.786672 -0.786890 -0.791303 -0.791998
Bas02  Non_Or -0.692721 -0.694442 -0.692735 -0.693719
Bas03  Non_Or -0.721510 -0.723214 -0.726308 -0.732325
Cel01  Non_Or -0.860155 -0.867227 -0.865590 -0.866864
Cel02  Non_Or -0.826620 -0.829116 -0.832455 -0.832014
```

```
[20]: data_new_nontarget.iloc[:5, :5]
```

```
[20]:      8998.367  8990.654  8982.939  8975.226  8967.512
      Name
Bas01 -0.786672 -0.786890 -0.791303 -0.791998 -0.799012
Bas02 -0.692721 -0.694442 -0.692735 -0.693719 -0.704448
Bas03 -0.721510 -0.723214 -0.726308 -0.732325 -0.735564
Cel01 -0.860155 -0.867227 -0.865590 -0.866864 -0.866571
Cel02 -0.826620 -0.829116 -0.832455 -0.832014 -0.833689
```

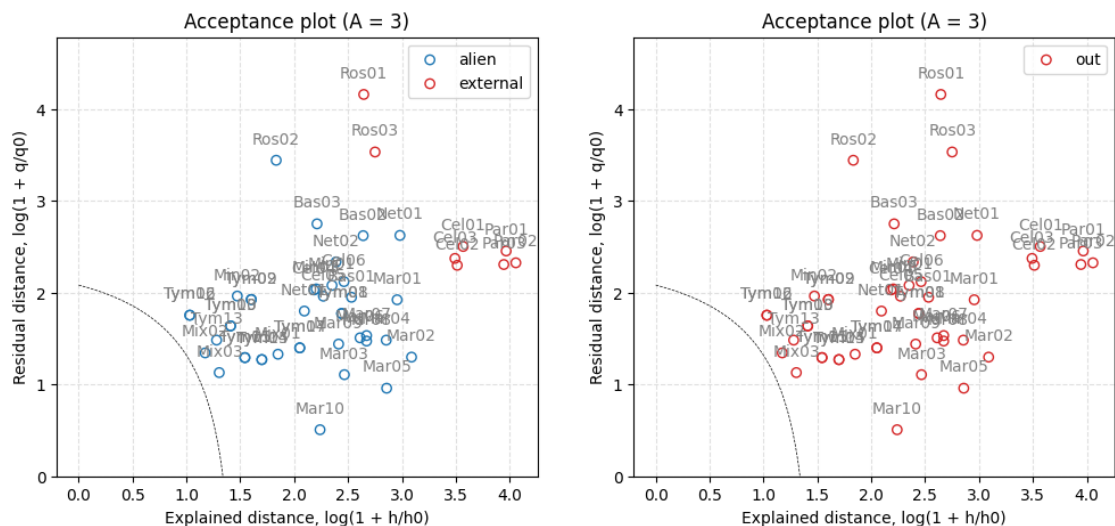
Now let's apply the model to both sets (remember that they contain the same objects, but one has a column with reference class labels and the other does not). Then check the acceptance plot:

```
[21]: r_test_nontarget = m_final.predict(data_test_nontarget)
      r_new_nontarget = m_final.predict(data_new_nontarget)

      plt.figure(figsize = (12, 5))

      ax1 = plt.subplot(1, 2, 1)
      r_test_nontarget.plotAcceptance(ax1, do_log = True, show_labels = True)

      ax2 = plt.subplot(1, 2, 2)
      r_new_nontarget.plotAcceptance(ax2, do_log = True, show_labels = True)
```



As you can see, in the first case the model indeed treated the objects as belonging to non-target classes and shows the corresponding roles (alien and external in this case) on the plot. In the second case, it simply splits the samples into accepted (in) and rejected (out).

Let's see how different the summary information is:

```
[22]: r_test_nontarget.summary()
```

DDSIMCA results:

- number of components (total): 10
- number of components (selected): 3
- limit type: classic
- alpha: 0.050
- gamma: 0.010

- class labels: provided
- number of objects: 49
- number of members: 0
- number of strangers: 49

PCs	eCrit	oCrit	in	out	TN	FP	spec	sel
1	12.592	26.238	5	44	44	5	0.898	0.900
2	15.507	30.124	0	49	49	0	1.000	0.969
3	14.067	28.215	0	49	49	0	1.000	0.968
4	15.507	30.124	0	49	49	0	1.000	0.984
5	16.919	31.977	0	49	49	0	1.000	0.989
6	16.919	31.977	0	49	49	0	1.000	0.984
7	22.362	38.986	0	49	49	0	1.000	0.995
8	22.362	38.986	0	49	49	0	1.000	0.994
9	27.587	45.558	0	49	49	0	1.000	0.999

```
10 31.410 50.294 0 49 49 0 1.000 1.000
```

```
[23]: r_new_nontarget.summary()
```

DDSIMCA results:

- number of components (total): 10
- number of components (selected): 3
- limit type: classic
- alpha: 0.050
- gamma: 0.010

- class labels: not provided
- number of objects: 49

PCs	eCrit	oCrit	in	out
1	12.592	26.238	5	44
2	15.507	30.124	0	49
3	14.067	28.215	0	49
4	15.507	30.124	0	49
5	16.919	31.977	0	49
6	16.919	31.977	0	49
7	22.362	38.986	0	49
8	22.362	38.986	0	49
9	27.587	45.558	0	49
10	31.410	50.294	0	49

The main difference is that for the second dataset there are no columns with figures of merit, true negatives, and false positives.

Now let's load data which has reference classes and contains objects of both target and non-target classes:

```
[24]: data_test_all = pd.read_csv("All_Test.csv", index_col = 0)
data_test_all.iloc[:5, :5]
```

```
[24]:      Class  8998.367  8990.654  8982.939  8975.226
Name
Bas01  Non_Or -0.786672 -0.786890 -0.791303 -0.791998
Bas02  Non_Or -0.692721 -0.694442 -0.692735 -0.693719
Bas03  Non_Or -0.721510 -0.723214 -0.726308 -0.732325
Cel01  Non_Or -0.860155 -0.867227 -0.865590 -0.866864
Cel02  Non_Or -0.826620 -0.829116 -0.832455 -0.832014
```

There are several differences here. First, the acceptance plot can now be shown only for members, only for strangers, or for all samples (the default option). In the latter case, the plot colors the points by class instead of by role until you change this by providing an explicit value for the parameter `show_set`:

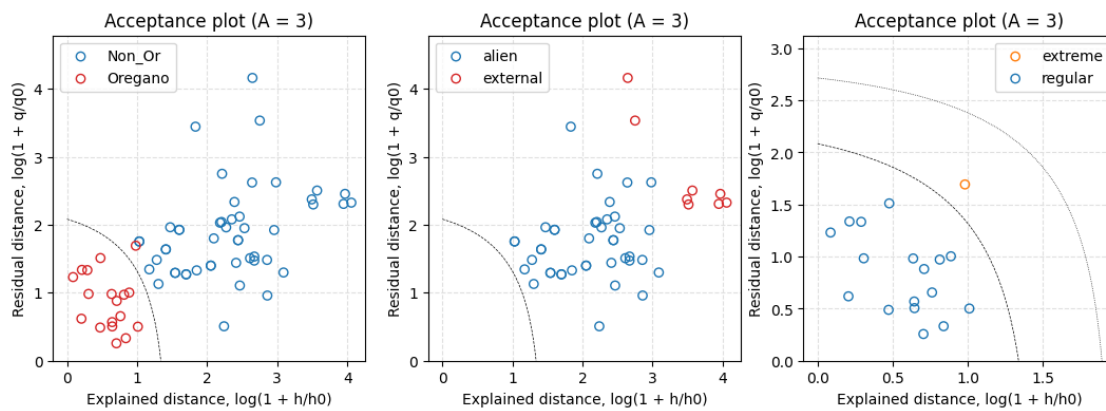
```
[25]: r_test_all = m_final.predict(data_test_all)

plt.figure(figsize = (13, 4))

ax1 = plt.subplot(1, 3, 1)
r_test_all.plotAcceptance(ax1, do_log = True)

ax2 = plt.subplot(1, 3, 2)
r_test_all.plotAcceptance(ax2, do_log = True, show_set = "strangers")

ax3 = plt.subplot(1, 3, 3)
r_test_all.plotAcceptance(ax3, do_log = True, show_set = "members")
```



You can provide your own colors, both for classification roles and for classes. Here are examples:

```
[26]: colors_classes = {"Oregano": "gray", "Non_Or": "magenta"}
markers_classes = {"Oregano": "s", "Non_Or": "o"}

colors_roles1 = {"external": "magenta", "alien": "green"}
markers_roles1 = {"external": "s", "alien": "o"}

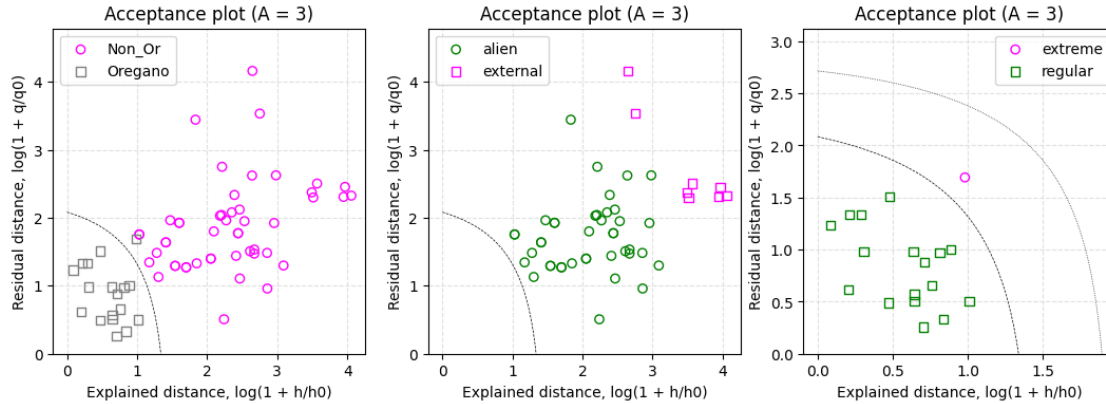
colors_roles2 = {"regular": "green", "extreme": "magenta", "outlier": "red"}
markers_roles2 = {"regular": "s", "extreme": "o", "outlier": "d"}

plt.figure(figsize = (13, 4))

ax1 = plt.subplot(1, 3, 1)
r_test_all.plotAcceptance(ax1, do_log = True, colors = colors_classes, markers_
    ⇨ markers_classes)

ax2 = plt.subplot(1, 3, 2)
r_test_all.plotAcceptance(ax2, do_log = True, show_set = "strangers",
    colors = colors_roles1, markers = markers_roles1)
```

```
ax3 = plt.subplot(1, 3, 3)
r_test_all.plotAcceptance(ax3, do_log = True, show_set = "members",
                           colors = colors_roles2, markers = markers_roles2)
```



Also, in this case all figures of merit, including accuracy and efficiency, are available:

```
[27]: r_test_all.summary()
```

DDSIMCA results:

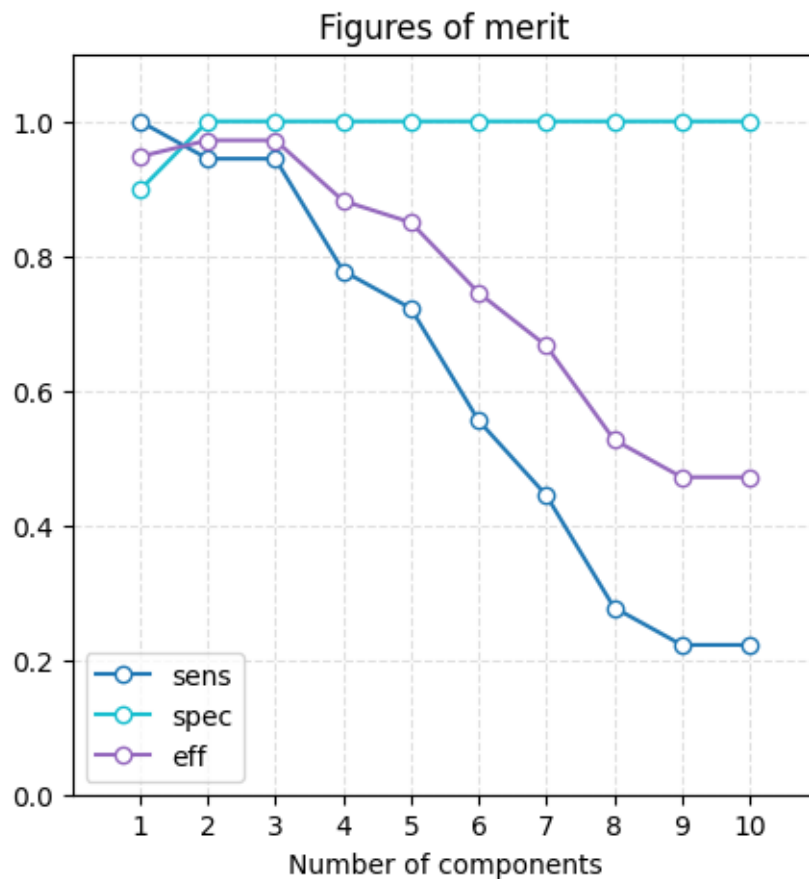
- number of components (total): 10
- number of components (selected): 3
- limit type: classic
- alpha: 0.050
- gamma: 0.010
- class labels: provided
- number of objects: 67
- number of members: 18
- number of strangers: 49

PCs	eCrit	oCrit	in	out	TP	FN	sens	TN	FP	spec	sel	acc	eff
1	12.592	26.238	23	44	18	0	1.000	44	5	0.898	0.900	0.925	0.948
2	15.507	30.124	17	50	17	1	0.944	49	0	1.000	0.969	0.985	0.972
3	14.067	28.215	17	50	17	1	0.944	49	0	1.000	0.968	0.985	0.972
4	15.507	30.124	14	53	14	4	0.778	49	0	1.000	0.984	0.940	0.882
5	16.919	31.977	13	54	13	5	0.722	49	0	1.000	0.989	0.925	0.850
6	16.919	31.977	10	57	10	8	0.556	49	0	1.000	0.984	0.881	0.745
7	22.362	38.986	8	59	8	10	0.444	49	0	1.000	0.995	0.851	0.667
8	22.362	38.986	5	62	5	13	0.278	49	0	1.000	0.994	0.806	0.527
9	27.587	45.558	4	63	4	14	0.222	49	0	1.000	0.999	0.791	0.471
10	31.410	50.294	4	63	4	14	0.222	49	0	1.000	1.000	0.791	0.471

And they can be plotted together:

```
[28]: plt.figure(figsize = (5, 5))

ax = plt.subplot(1, 1, 1)
r_test_all.plotFoM(ax, fom = "sens")
r_test_all.plotFoM(ax, fom = "spec")
r_test_all.plotFoM(ax, fom = "eff")
```



1.4 Extra plots, features, and details

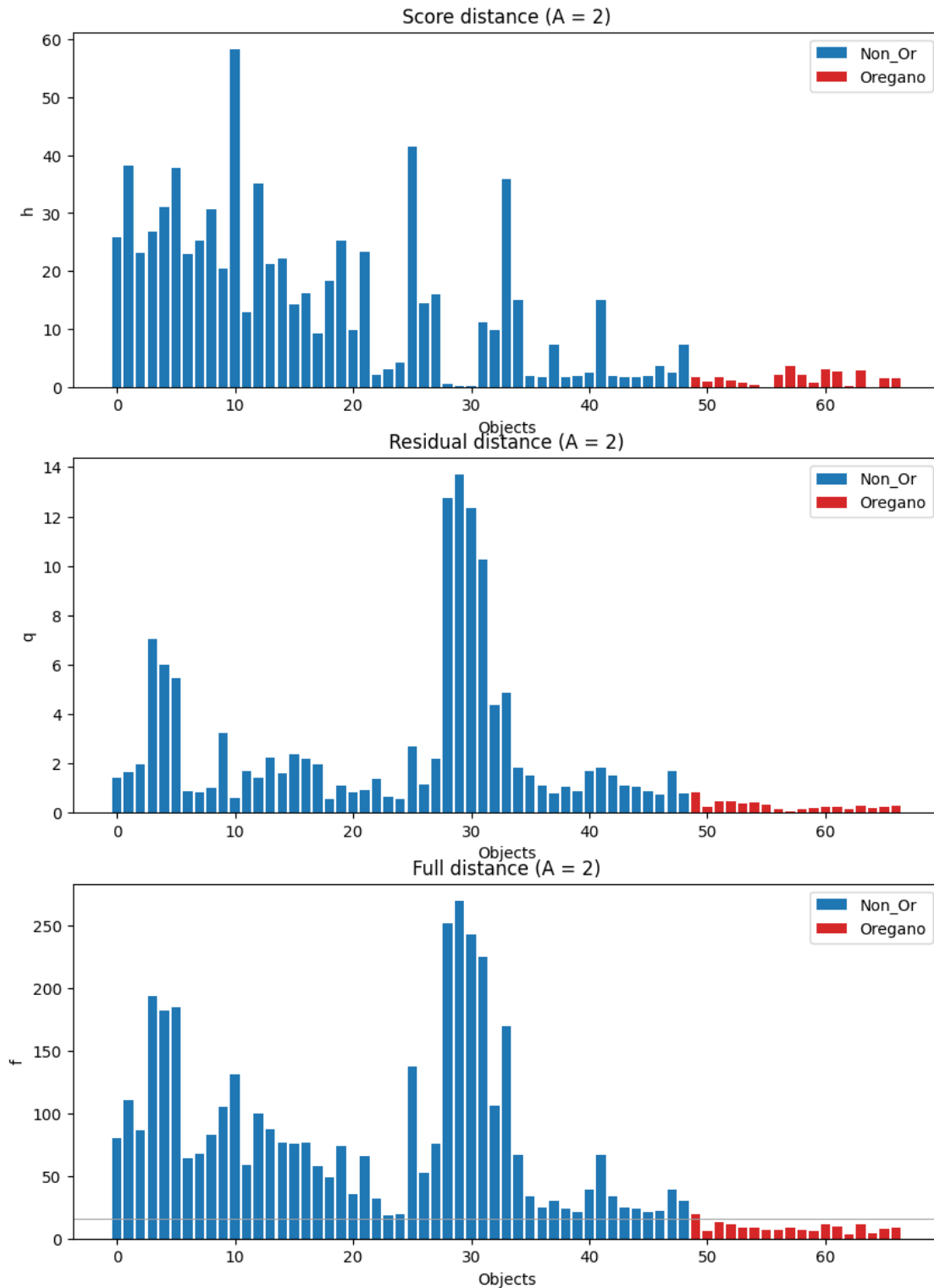
It is also possible to show every distance separately for a given number of components (this works for any result object):

```
[29]: plt.figure(figsize = (10, 14))

ax1 = plt.subplot(3, 1, 1)
r_test_all.plotDistance(ax1, ncomp = 2, distance = "h")
```

```
ax2 = plt.subplot(3, 1, 2)
r_test_all.plotDistance(ax2, ncomp = 2, distance = "q")

ax3 = plt.subplot(3, 1, 3)
r_test_all.plotDistance(ax3, ncomp = 2, distance = "f", show_crit = True)
```



Similar to the acceptance plot, you can provide your own colors for each class as a dictionary. You can also show the log-transformed distances.

```

[30]: colors = {"Oregano": "cyan", "Non_Or": "pink"}

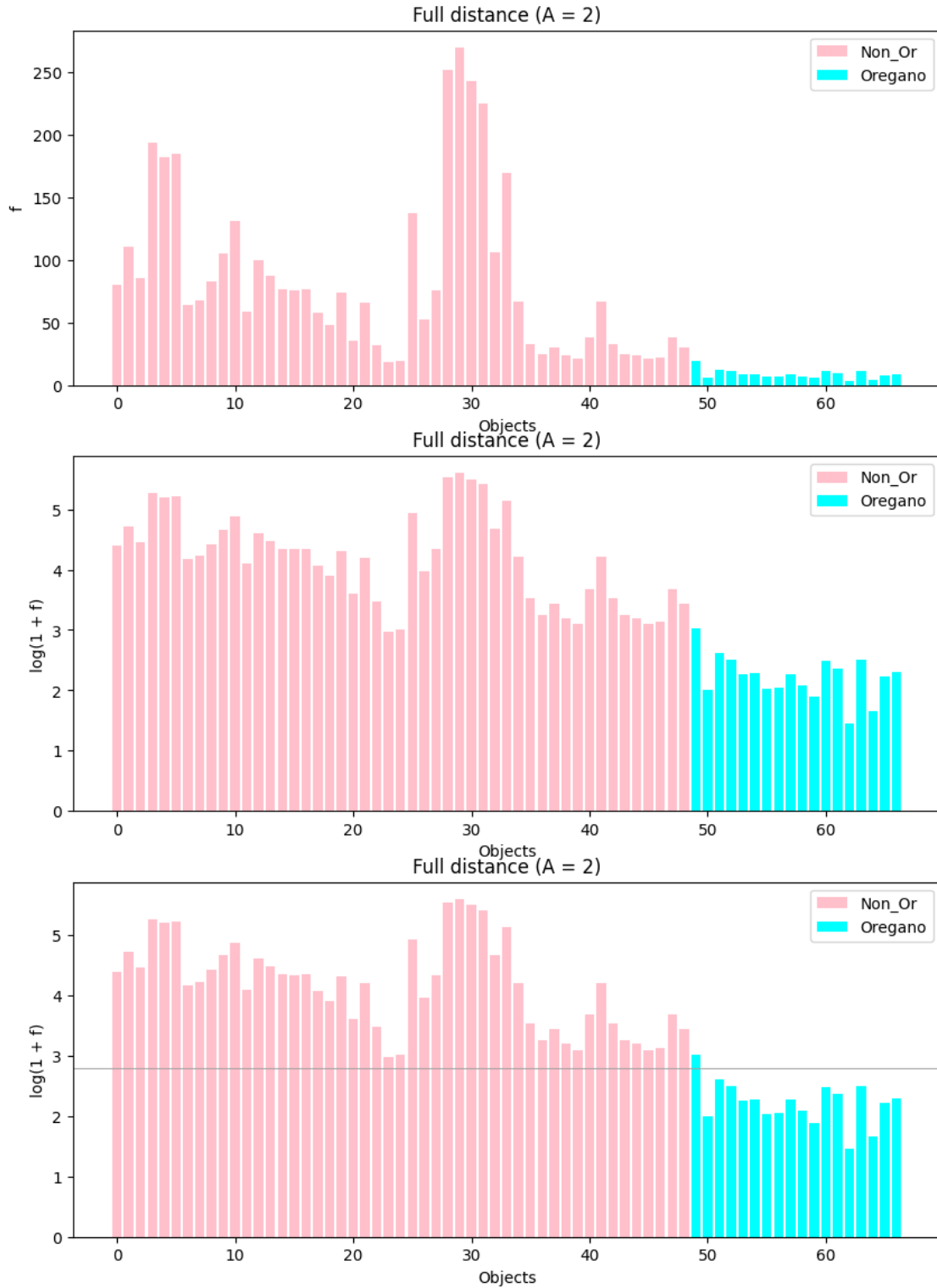
plt.figure(figsize = (10, 14))

ax1 = plt.subplot(3, 1, 1)
r_test_all.plotDistance(ax1, ncomp = 2, distance = "f", colors = colors)

ax2 = plt.subplot(3, 1, 2)
r_test_all.plotDistance(ax2, ncomp = 2, distance = "f", colors = colors,
    ↪do_log=True)

ax3 = plt.subplot(3, 1, 3)
r_test_all.plotDistance(ax3, ncomp = 2, distance = "f", colors = colors,
    ↪do_log=True, show_crit = True)

```

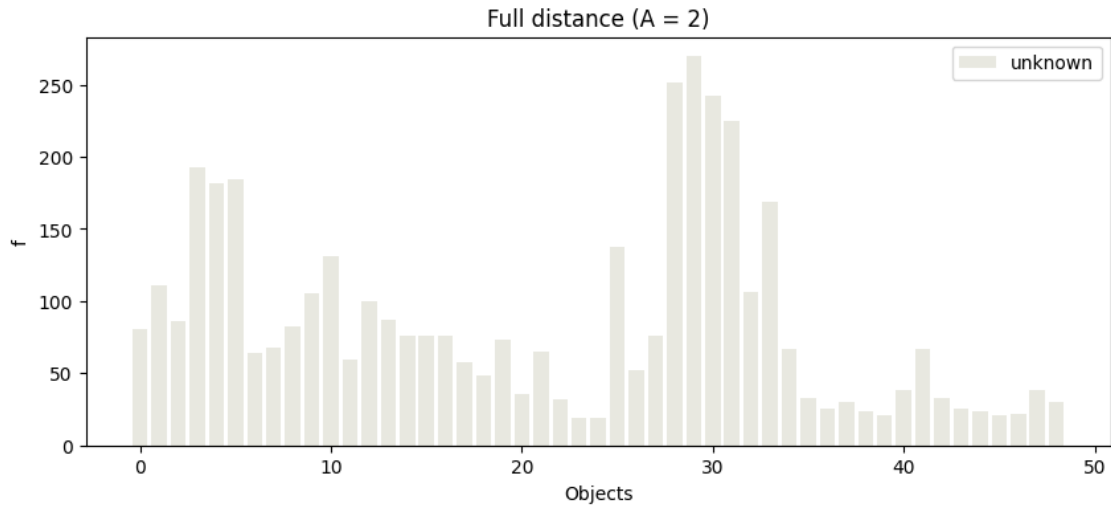


If you want to change the color for results obtained from new data without class labels, use the label "unknown" as shown below. Make sure the labels match the class names exactly, including

case — otherwise you will get an error.

```
[31]: colors = {"unknown": "#e8e8e0"}

plt.figure(figsize = (10, 4))
ax = plt.subplot(1, 1, 1)
r_new_nontarget.plotDistance(ax, ncomp = 2, distance = "f", colors = colors)
```



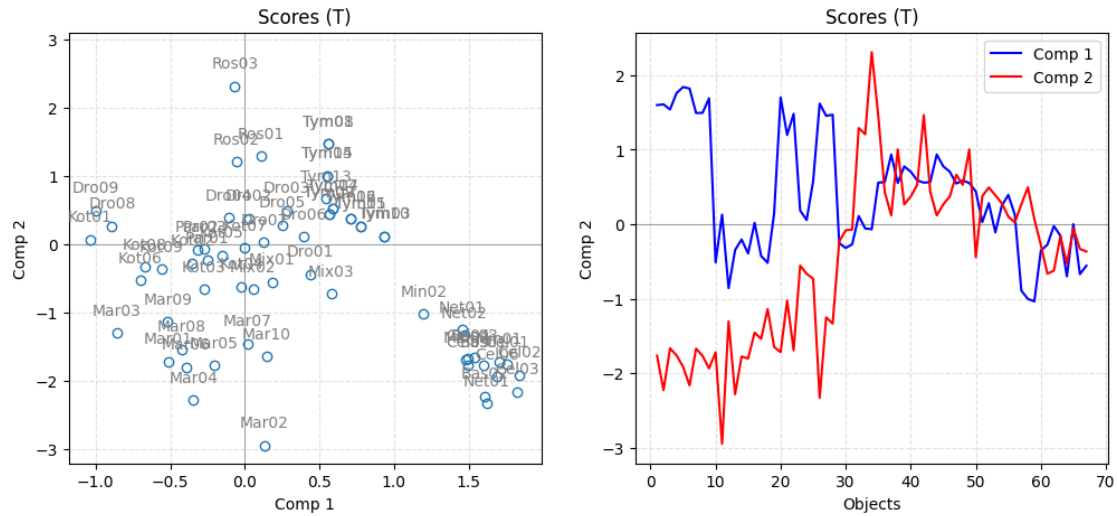
You can also show the PCA scores plot:

```
[32]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_test_all.plotScores(ax1, comp = (1, 2), type = "p", show_labels = True)

ax2 = plt.subplot(1, 2, 2)
r_test_all.plotScores(ax2, comp = (1,), type = "l", color = "blue")
r_test_all.plotScores(ax2, comp = (2,), type = "l", color = "red")
ax2.legend()
```

```
[32]: <matplotlib.legend.Legend at 0x11e67b230>
```



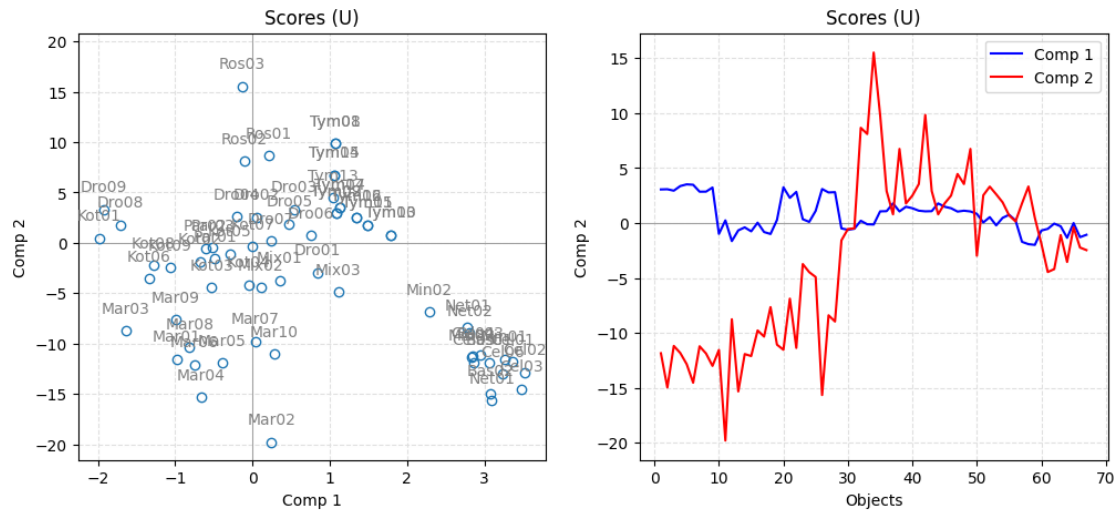
By default, T scores are shown. If you want to show the standardized scores (also known as left singular vectors), U, just provide an extra argument:

```
[33]: plt.figure(figsize = (12, 5))

ax1 = plt.subplot(1, 2, 1)
r_test_all.plotScores(ax1, comp = (1, 2), type = "p", scores = "U", show_labels_
    ↪= True)

ax2 = plt.subplot(1, 2, 2)
r_test_all.plotScores(ax2, comp = (1,), type = "l", scores = "U", color = _
    ↪"blue")
r_test_all.plotScores(ax2, comp = (2,), type = "l", scores = "U", color = "red")
ax2.legend()
```

```
[33]: <matplotlib.legend.Legend at 0x11e4292b0>
```



Similar to the web application, you can also make a plot with expected vs. observed alien objects and a selectivity vs. sensitivity plot (if there are non-target class objects in the dataset):

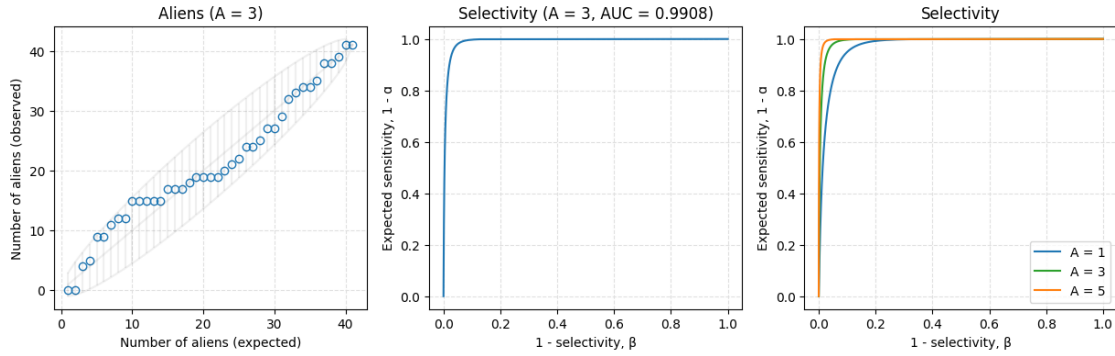
```
[34]: plt.figure(figsize = (15, 4))

ax1 = plt.subplot(1, 3, 1)
r_test_all.plotAliens(ax1)

ax2 = plt.subplot(1, 3, 2)
r_test_all.plotSelectivity(ax2)

ax2 = plt.subplot(1, 3, 3)
r_test_all.plotSelectivity(ax2, ncomp = 1, color = "tab:blue", label = "A = 1")
r_test_all.plotSelectivity(ax2, ncomp = 3, color = "tab:green", label = "A = 3")
r_test_all.plotSelectivity(ax2, ncomp = 5, color = "tab:orange", label = "A = 5")
ax2.set_title("Selectivity")
ax2.legend()
```

```
[34]: <matplotlib.legend.Legend at 0x11e7be3c0>
```

Like the distance, extremes, and acceptance plots, the plots above are made for the optimal number of components we pre-selected earlier ($A = 3$). You can change this by providing a value via the `ncomp` argument to the plotting methods.

Any result object can also be converted to a data frame, where every object is represented by several columns: reference class (if provided), decision (in / out), role, and distance values. By default, these values are computed for the optimal number of components, but you can specify which number of components to use for creating the data frame:

```
[35]: rdf = r_test_all.as_df(ncomp = 2)
      rdf.head()
```

```
[35]:
```

	class	decision	role	h	q	f
Bas01	Non_Or	out	alien	25.784223	1.404320	80.211486
Bas02	Non_Or	out	external	38.279119	1.663383	110.801080
Bas03	Non_Or	out	alien	23.142295	1.967705	85.888616
Cel01	Non_Or	out	external	26.719913	7.047244	192.987414
Cel02	Non_Or	out	external	31.019882	6.011383	181.411337

You can also access all computed values directly. For a model, this includes loadings, vectors for centering and scaling, classic and robust parameters of the distance distributions (for the h -, q -, and f -distances), and more. Here are some examples:

```
[36]: # loadings
      m_final.V[:5, :5]
```

```
[36]: array([[ 0.0154751 , -0.07436178,  0.00604071, -0.10195844,  0.00927309],
             [ 0.0147401 , -0.07253297,  0.00536661, -0.09940793,  0.00525554 ],
             [ 0.01478209, -0.07205679,  0.00486695, -0.09530707,  0.00452151],
             [ 0.01524467, -0.07206864,  0.00411975, -0.09351868,  0.00561563],
             [ 0.01508584, -0.07140133,  0.00455854, -0.09364199,  0.00728575]])
```

The distance parameters are saved as dictionaries `hParams`, `qParams`, and `fParams`. Each dictionary has two fields, `classic` and `robust`. Each field contains a tuple with scaling factors (e.g. `h0`) and the number of degrees of freedom (e.g. `Nh`) computed for each component in the model. Here is an example for score distance parameters computed using classic estimates:

```
[37]: m_final.hParams["classic"]
```

```
[37]: (array([0.98, 1.96, 2.94, 3.92, 4.9 , 5.88, 6.86, 7.84, 8.82, 9.8 ]),
      array([3., 4., 5., 5., 6., 5., 6., 7., 7., 9.]))
```

Here is another example showing how to get q_0 and N_q values for the robust estimator and 2 PCs:

```
[38]: q0, Nq = m_final.qParams["robust"]
      A = 2
      print(f"A = {A}: q0 = {q0[A - 1]:.3f}, Nq = {Nq[A - 1]}")
```

A = 2: $q_0 = 0.203$, $N_q = 6.0$

For a result object, you can access all outcomes — including critical values, distances, calculations related to the Type II error (if non-target class objects are provided), and many other things — via the `outcomes` data frame:

```
[39]: r_test_all.outcomes.head()
```

```
[39]:   PCs      eCrit      oCrit  TP  FN  TN  FP      beta      s      f0t  \
0    1  12.591587  26.238401  18   0  44   5  0.100312  1.809481  4.276287
1    2  15.507313  30.123804  17   1  49   0  0.030801  0.475361  6.338419
2    3  14.067140  28.215317  17   1  49   0  0.031760  1.936770  5.942245
3    4  15.507313  30.123804  14   4  49   0  0.016483  6.039777  4.142575
4    5  16.918978  31.976664  13   5  49   0  0.010570  3.064697  5.896735

      ...      Sz      k      m  in  out      sens      spec      sel  \
0  ...  0.201182  6.0  33.395585  23  44  1.000000  0.897959  0.899688
1  ...  0.167382  8.0  53.720391  17  50  0.944444  1.000000  0.969199
2  ...  0.185492  7.0  53.104481  17  50  0.944444  1.000000  0.968240
3  ...  0.178675  8.0  58.160822  14  53  0.777778  1.000000  0.983517
4  ...  0.164901  9.0  71.142327  13  54  0.722222  1.000000  0.989430

      acc      eff
0  0.925373  0.947607
1  0.985075  0.971825
2  0.985075  0.971825
3  0.940299  0.881917
4  0.925373  0.849837
```

[5 rows x 22 columns]

You can also access matrices (`nrows x ncomp`) with the h -, q -, and f -distances, and a matrix with decisions:

```
[40]: r_test_all.H[:3, :5]
```

```
[40]: array([[ 4.89404283, 25.78422342, 34.04616547, 36.75323386, 49.57135654],
            [ 4.95295747, 38.27911861, 38.28482731, 52.63494504, 60.14293327],
```

```
[ 4.54079899, 23.14229481, 23.92305847, 41.62511717, 42.63880225]])
```

```
[41]: r_test_all.Q[:3, :5]
```

```
[41]: array([[4.51563465, 1.40431952, 0.78597217, 0.58591372, 0.27215077],
          [6.62687216, 1.66338278, 1.66295553, 0.60244953, 0.41866844],
          [4.73815108, 1.96770517, 1.90927059, 0.60104861, 0.57623555]])
```

```
[42]: r_test_all.F[:3, :5]
```

```
[42]: array([[ 53.73683932,  80.21148642,  69.97052161,  77.27749275,
             84.83000905],
          [ 72.0367184 , 110.80108046,  90.64551871,  98.39268782,
            110.76586145],
          [ 54.56521024,  85.88861591,  70.00299634,  84.27685746,
            103.30300075]])
```

```
[43]: r_test_all.D[:3, :5]
```

```
[43]: array([[False, False, False, False, False],
          [False, False, False, False, False],
          [False, False, False, False, False]])
```

Here, for example, are the decisions obtained for each object using $A = 3$ components:

```
[44]: r_test_all.D[:, 2]
```

```
[44]: array([False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, False, False, False, False, False,
          False, False, False, False, False, True, True, True, True,
          True, True, True, True, True, True, True, True, True, True,
          True, True, True, True])
```

And a matrix with roles, which are coded as integer values: 0 for **regular**, 1 for **extreme**, 2 for **outlier**, 3 for **alien**, and 4 for **external**. The first three are assigned to target class members; the last two — to objects from non-target classes and to unknown objects:

```
[45]: r_test_all.R[:, 2]
```

```
[45]: array([3, 3, 3, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
          3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 3, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
          3, 3, 3, 3, 3, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
          0], dtype=int16)
```

Here is a simple way to get the role names only for members:

```
[46]: import numpy as np
r_names = np.array(["regular", "extreme", "outlier", "alien", "external"])
r_names[r_test_all.R[:, 2]]
```

```
[46]: array(['alien', 'alien', 'alien', 'external', 'external', 'external',
            'alien', 'alien', 'alien', 'alien', 'alien', 'alien', 'alien',
            'alien', 'alien', 'alien', 'alien', 'alien', 'alien', 'alien',
            'alien', 'external', 'external', 'external', 'external', 'alien',
            'external', 'alien', 'alien', 'alien', 'alien', 'alien', 'alien',
            'alien', 'alien', 'alien', 'alien', 'alien', 'alien', 'alien',
            'alien', 'alien', 'extreme', 'regular', 'regular', 'regular',
            'regular', 'regular', 'regular', 'regular', 'regular', 'regular',
            'regular', 'regular', 'regular', 'regular', 'regular', 'regular',
            'regular', 'regular'], dtype='<U8')
```