mclust

September 3, 2018

1 Mclust

The mclust package uses NumPy arrays for most inputs and outputs. Therefore numpy should be imported to set up the data. In this example the diabetes data set inside the mclust module is used, so pkg_resources is also loaded, to load the data.

```
In [1]: import numpy as np
        import pkg_resources
```

The diabetes data set consists of the information about the glucose, insulin and sspg levels of 145 individuals. Classifications for the individuals are also available, but are not considered in this clustering example. The data can be loaded as follows:

```
In [2]: resource_path = 'resources/data_sets/diabetes.csv'
        with pkg_resources.resource_stream('mclust', resource_path) as f:
            diabetes = np.genfromtxt(f, delimiter=',', skip_header=1)
```

The main functionality of mclust is clustering. The Mclust class provides the basic functionality for clustering. Internally it fits all available model configurations for the data on 1 to 9 groups, by default. The Mclust class is available in the clustering module, and can be imported with

```
In [3]: from mclust.clustering import Mclust, Model
```

The `Model` class is also imported. This class is an enumartion that lists all model configurations. It is used later in this chapter to run mclust with a limit set of model configurations. `Model` does not have to be imported if `Mclust` is used with default settings.

The Mclust class has one required constructor parameter, the data set to fit the clustering model on. The data set has to be presented as a numpy array. The `Mclust` object can be initialised as follows:

```
In [4]: model = Mclust(diabetes)
```

This sets up an Mclust object that runs all available models for the given data on 1 to 9 clusters. For multidimensional data with more observations than parameters all 14 model configurations are available, if there are more parameters than observations only EEE, EEV, VEV and VVV are available. For one dimensional data 2 models are available, namely, E and V.

The mclust object can be fitted to the data, by simply calling fit on it, this may take some time, depending on the dimensions of the data. The fit function returns an integer indicating the success status of the model. If the model ran correctly, the return value will be 0.

```
In [5]: model.fit()
```

```
Out[5]: 0
```

A summary of the fitted model can be printed, by just printing the Mclust object.

```
In [6]: print(model)
```

```
modelname: Model.VVV
n: 145
d: 3
g: 3
mean:
[[  90.96612189  357.83541135  163.77925628]
 [ 104.57491007  495.15787528  309.44078639]
 [ 229.71014968 1099.58435216   81.45235717]]
variance:
[[[ 5.71925559e+01  7.61249242e+01  1.48098662e+01]
  [ 7.61249242e+01  2.10527688e+03  3.24038682e+02]
  [ 1.48098662e+01  3.24038682e+02  2.41981204e+03]]

 [[ 1.85859476e+02  1.28584836e+03 -5.15919177e+02]
  [ 1.28584836e+03  1.40552439e+04 -2.60695388e+03]
  [-5.15919177e+02 -2.60695388e+03  2.38598523e+04]]

 [[ 5.50978995e+03  2.02874794e+04 -2.47560276e+03]
  [ 2.02874794e+04  8.26297140e+04 -1.03419499e+04]
  [-2.47560276e+03 -1.03419499e+04  2.21278128e+03]]]
pro: [0.53738994 0.26502765 0.19758241]
loglik: -2303.4955606441354
return_code: 0
```

The best clustering model for the diabetes data set is a model where the clusters have varying volume, varying shape and a varying orientation. The resulting mixture model has 3 cluster components. Roughly 53.7% of the data belongs to cluster 1, 26.5% to cluster 2 and 19.8% to cluster 3. Furthermore the output specifies the mean and covariance matrices for all 3 cluster components.

The BIC value, which is used for comparing the models can be obtained by calling

```
In [7]: model.bic()
```

```
Out[7]: -4751.316399818467
```

The cluster assignments for the data used to fit the model is given by calling the `predict` function. This results in a vector where the indices of the elements correspond to the indices of the original data. The elements can take the values 0 to g-1, where each number indicates to which cluster the data point is assigned.

```
In [8]: model.predict()
```

```
Out[8]: array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
               0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1,
               1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 2, 1, 1, 0,
               1, 1, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2,
               2, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2])
```

The predict function can also be used to assign new data points to the clusters present in the model. Note that the new data should be a NumPy array. Again the indices of the resulting array correspond to the indices of the input data.

```
In [9]: new_data = np.array([[200, 300, 80],
                             [100, 500, 150]])
        model.predict(new_data)

Out[9]: array([2, 1])
```

Furthermore the density of a data point can be calculated. There are two options for this. The density of the data point in each component of the mixture model, and the density in the model as a whole can be computed. The `component_density` and `density` functions of a mixture model can be used for this, respectively. Both functions behave in a similar manner to the `predict` function, they can be used on new observations or existing observations. If no new data is supplied, the densities for the training data are computed.

```
In [10]: # compute component densities
         print(model.component_density()[0:5]) # only print first 5 observations
         print(model.component_density(new_data))

[[9.39801573e-07 1.72068603e-08 7.92516900e-10]
 [4.83153378e-07 1.74950901e-08 4.66868720e-12]
 [2.86428081e-07 1.31292981e-08 3.93990391e-11]
 [2.92653681e-06 1.33816336e-07 1.60856687e-09]
 [7.08964639e-07 1.21572753e-07 1.33169315e-10]]
[[2.40139247e-56 6.84284138e-49 7.94000085e-24]
 [2.10232017e-08 1.75852846e-07 2.59810245e-08]]
```

```
In [11]: # compute model densities
         print(model.density()[0:10]) # only print first 5 observations
         print(model.density(new_data))

[5.09756792e-07 2.64279370e-07 1.57410981e-07 1.60847430e-06
 4.13236917e-07 1.32759781e-06 4.80322586e-07 6.96167837e-07
 1.29308860e-06 4.01915396e-07]
[1.56880453e-24 6.30369165e-08]
```

Sometimes only a selection of available models and cluster components might be of interest, for example only models with spherical components on 1 to 3 clusters. This can be specified in the Mclust constructor by using the `models` and `groups` parameters. Note that the `Model` class needs to be imported to specify which models to use.

```
In [12]: model2 = Mclust(diabetes, models=[Model.EII, Model.VII],
                         groups=range(1,4))
         model2.fit()
         print(model2)
```

```
modelname: Model.VII
n: 145
d: 3
g: 3
mean:
[[  91.06215155  355.617936    159.55491577]
 [ 104.61526852  500.06535618  298.39767233]
 [ 242.46611462 1155.51073614   75.18901052]]
variance:
[[[ 1363.29582503     0.             0.        ]
  [    0.          1363.29582503     0.        ]
  [    0.             0.          1363.29582503]]

 [[12240.19635635     0.             0.        ]
  [    0.         12240.19635635     0.        ]
  [    0.             0.         12240.19635635]]

 [[22312.35127108     0.             0.        ]
  [    0.         22312.35127108     0.        ]
  [    0.             0.         22312.35127108]]]
pro: [0.52338063 0.29914946 0.17746991]
loglik: -2568.360249242583
return_code: 0
```

```
/home/joris/miniconda3/envs/cleannumpy/lib/python3.6/site-packages/mclust/clustering.py:45: Us
  warnings.warn("optimal number of clusters occurs at max choice")
```

This code raises a warning. The warning suggest that there might be a better clustering possible outside the range of cluster components specified. All warning can be disable using the following code

```
In [13]: import warnings
         warnings.filterwarnings('ignore')
```

Warnings can be enabled again by running

```
In [14]: warnings.filterwarnings('default')
```

2 ModelFactory

It is also possible to just use a single model configuration. The ModelFactory class can be used for this purpose. This class is available in the model_factory module.

```
In [15]: from mclust.model_factory import ModelFactory
```

The `ModelFactory` class has a static method `create` that sets up and returns a mixture model. This function requires 2 parameters, the data to fit the model on, and the model configuration to be fitted. Furthermore an initial clustering can be supplied via the z parameter, or the number of groups can be specified. If the number of groups is specified, hierarchical clustering will be used to initialise the cluster assignment.

```
In [16]: model3 = ModelFactory.create(diabetes, Model.VVV, groups=3)
         model3.fit()
         print(model3)
```

```
modelname: Model.VVV
n: 145
d: 3
g: 3
mean:
[[  90.96612189  357.83541135  163.77925628]
 [ 104.57491007  495.15787528  309.44078639]
 [ 229.71014968 1099.58435216   81.45235717]]
variance:
[[[ 5.71925559e+01  7.61249242e+01  1.48098662e+01]
  [ 7.61249242e+01  2.10527688e+03  3.24038682e+02]
  [ 1.48098662e+01  3.24038682e+02  2.41981204e+03]]

 [[ 1.85859476e+02  1.28584836e+03 -5.15919177e+02]
  [ 1.28584836e+03  1.40552439e+04 -2.60695388e+03]
  [-5.15919177e+02 -2.60695388e+03  2.38598523e+04]]

 [[ 5.50978995e+03  2.02874794e+04 -2.47560276e+03]
  [ 2.02874794e+04  8.26297140e+04 -1.03419499e+04]
  [-2.47560276e+03 -1.03419499e+04  2.21278128e+03]]]
pro: [0.53738994 0.26502765 0.19758241]
loglik: -2303.4955606441354
return_code: 0
```

The user can also supply an own initial clustering, by using the z parameter instead of groups. An example is shown below

```
In [17]: z_init = np.random.multinomial(1, [.25, .25, .25, .25], 145)
         # z matrix must be fortran contiguous and float type
         z_init = np.asfortranarray(z_init, float)
         model4 = ModelFactory.create(diabetes, Model.VVV, z=z_init)
         model4.fit()

         print(model4)
```

```
modelname: Model.VVV
n: 145
d: 3
g: 4
mean:
[[ 100.59046928  477.31410186  389.86516999]
 [ 306.19421768 1387.5540353    49.84546866]
 [  91.70927063  358.72762043  168.51262115]
 [ 150.56352902  777.55970351  144.34246335]]
variance:
[[[ 1.06730277e+02  4.31864157e+02  3.31886890e+02]
  [ 4.31864157e+02  2.76622141e+03  3.39333675e+03]
  [ 3.31886890e+02  3.39333675e+03  2.29499885e+04]]

 [[ 1.00140979e+03  2.01774927e+03 -5.62354077e+02]
  [ 2.01774927e+03  1.51005162e+04 -3.85292827e+03]
  [-5.62354077e+02 -3.85292827e+03  1.39312258e+03]]

 [[ 6.85111998e+01  2.08425378e+02 -1.15356550e+01]
  [ 2.08425378e+02  3.77961211e+03 -5.76605206e+02]
  [-1.15356550e+01 -5.76605206e+02  3.26052655e+03]]

 [[ 1.79689571e+03  8.01490501e+03 -2.11616217e+03]
  [ 8.01490501e+03  3.88769449e+04 -9.31641893e+03]
  [-2.11616217e+03 -9.31641893e+03  5.89861413e+03]]]
pro: [0.14482262 0.08326143 0.58276372 0.18915223]
loglik: -2302.7414344235685
return_code: 0
```

The interface for all models is the same. Therefor the model created by the ModelFactory can be accessed in the same way as the model resulting from the Mclust objects, independent on the model configuration that was used. A few examples:

```
In [18]: model3.predict(new_data)

Out[18]: array([2, 1])

In [19]: model_list = [model, model2, model3, model4]
         for m in model_list:
             print(m.bic())

-4751.316399818467
-5206.394770879054
-4751.316399818467
-4799.5754848015395
```

3 MclustBIC

Sometimes the model selected using BIC values might not be significantly better than other models. The `MclustBIC` class can be used to fit multiple models with multiple different cluster components. It can be imported by running

```
In [20]: from mclust.clustering import MclustBIC
```

The constructor for `MclustBIC` has a similar interface to the `Mclust` class. It has one mandatory parameter, namely, the data to fit the models on. Like `Mclust` there are also parameters that allow to only fit a selection of models on a limit amount of cluster components. The basic call to `MclustBIC` is as follows:

```
In [21]: bic = MclustBIC(diabetes)
```

This command directly fits all possible models on 1 to 9 cluster components by default. The return codes for all models fitted can be viewed by calling

```
In [22]: bic.get_return_codes_matrix()

Out[22]: array([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
               [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
```

The result of this call is a matrix where the `[i, j]`th element indicates the return code of model configuration `bic.models[j]` with `bic.groups[j]` cluster components. A return code of 0 indicates that the model was fitted correctly.

The BIC values for all models can be accessed by the `get_bic_matrix` function. This returns a similarly structured matrix to the return code matrix, except that the elements indicate the BIC of model configuration `bic.models[j]` with `bic.groups[i]` cluster components.

```
In [23]: # Only print with one digit precision.
         np.set_printoptions(precision = 1)
         bic.get_bic_matrix()

Out[23]: array([[-5863.9, -5863.9, -5530.1, -5530.1, -5530.1, -5530.1, -5136.4,
                -5136.4, -5136.4, -5136.4, -5136.4, -5136.4, -5136.4, -5136.4],
               [-5449.5, -5327.8, -5169.4, -5019.4, -5015.9, -4988.3, -5011. ,
                -4875.6, -4920.4, -4877.1, -4918.5, -4834.7, -4823.8, -4825. ],
               [-5412.6, -5206.4, -4998.4, -4899.8, -4920.5, -4827.8, -4976.8,
                -4830.5, -4874. , -4840. , -4861.2, -4809.1, -4817.6, -4751.3],
               [-5236. , -5193.2, -4937.6, -4902.9, -4865.8, -4813. , -4865.9,
                -4849. , -4856.7, -4792.6, -4874.1, -4818.5, -4827. , -4784.3],
               [-5181.6, -5125.7, -4934.3, -4836.7, -4838.7, -4820.7, -4858.2,
                -4787.9, -4822.2, -4814.7, -4896.5, -4837.9, -4812. , -4804.2],
               [-5162.2, -5114.3, -4886.3, -4824.3, -4840.4, -4827.1, -4848.5,
                -4785.2, -4805.7, -4789.6, -4842.1, -4833.6, -4855.1, -4830.8],
               [-5181.3, -5110.3, -4906.2, -4849.8, -4865. , -4853.6, -4868.5,
                -4787.3, -4815.8, -4818. , -4871.2, -4855.8, -4844.7, -4858.6],
               [-5153.7, -5091.5, -4898.8, -4855.8, -4868.6, -4881.8, -4866.6,
                -4809.1, -4827.5, -4821.2, -4877.1, -4887. , -4854.3, -4907.5],
               [-5102.3, -5095.9, -4879. , -4860.1, -4913.6, -4901. , -4871. ,
                -4843.8, -4847.4, -4874.6, -4905.7, -4912.9, -4914.4, -4948.8]])
```

All fitted models are accessible, via the `fitted_models` field. This is a dictionary that takes the number of cluster components together with the model configuration as the keys. For example, the mean of the EEE model with 1 cluster component can be accessed in the following way.

```
In [24]: print(bic.fitted_models[1, Model.EEE].mean)
```

```
[[122.  540.8 186.1]]
```

A model can be selected based on its index in the BIC matrix as follows:

```
In [25]: print(bic.get_bic_matrix()[1, 6])
         print(bic.models[6])
         print(bic.groups[1])
         print(bic.fitted_models[bic.groups[1], bic.models[6]])
```

```
-5010.985852685136
Model.EEE
2
modelname: Model.EEE
n: 145
d: 3
g: 2
mean:
[[ 101.2  442.3  203.1]
 [ 281.  1295.9   55.7]]
variance:
[[[  760.7  3878.6  -337.6]
  [ 3878.6 27011.6  -564.7]
  [ -337.6  -564.7 12306.4]]

 [[  760.7  3878.6  -337.6]
  [ 3878.6 27011.6  -564.7]
  [ -337.6  -564.7 12306.4]]]
pro: [0.9 0.1]
loglik: -2473.144157016834
return_code: 0
```

Like the `Mclust` class, also a limited number of model configurations and cluster components can be selected. This is done by supplying the `groups` and/or `models` parameters.

```
In [26]: bic2 = MclustBIC(diabetes, groups=[2, 4], models=[Model.VVV, Model.VEV])
         bic2.get_bic_matrix()

Out[26]: array([[-4825. , -4834.7],
                [-4784.3, -4818.5]])
```

4 Discriminant Analysis

The mclust package also supports discriminant analysis. There are two methods for discriminant analysis available, namely, `EDDA` and `MclustDA`. Both are availbable from the `classification` module.

```
In [27]: from mclust.classification import EDDA, MclustDA
```

For classification the labels of the training data should be known a priori. The classification labels for the diabetes data can be loaded with the following code

```
In [28]: resource_path = 'resources/data_sets/diabetes_classification.csv'
         with pkg_resources.resource_stream('mclust', resource_path) as f:
             classes = np.genfromtxt(f, delimiter=',')
```

In this exampke the diabetes data is split in a training and test set.

```
In [29]: training_ind = np.random.choice(range(145), 100, replace=False)
         mask = np.ones(len(diabetes), np.bool)
         mask[training_ind] = 0

         training = diabetes[training_ind]
         training_labels = classes[training_ind]

         test = diabetes[mask]
         test_labels = classes[mask]
```

The `EDDA` class can be set up as follows

```
In [30]: da = EDDA(training, training_labels)
```

The predicted classes for the training data can be obtained by calling the `predict` method without any parameters.

```
In [31]: da.predict()

Out[31]: array([2, 1, 1, 0, 2, 1, 1, 0, 1, 1, 1, 2, 0, 0, 0, 1, 2, 1, 1, 1, 1, 1,
                2, 1, 0, 2, 1, 0, 1, 1, 0, 1, 1, 2, 1, 2, 0, 1, 1, 1, 1, 1, 0, 1,
                1, 1, 1, 1, 2, 1, 1, 1, 1, 0, 1, 1, 1, 2, 2, 0, 0, 2, 2, 0, 1, 2,
                1, 0, 1, 0, 1, 0, 1, 0, 1, 2, 0, 1, 1, 2, 1, 1, 0, 1, 2, 2, 1, 0,
                0, 1, 2, 1, 1, 2, 1, 1, 2, 2, 2, 1])
```

By passing the test data to the `predict` function, the predicted labels for the test data can be obtained.

```
In [32]: pred = da.predict(test)
         pred

Out[32]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
                1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 2, 2, 2, 2, 2, 2, 2,
                2])
```

With the following helper functions, the confusion matrix of the can be computed and displayed. The confusion matrix gives an indication about how well the class predictions are.

```
In [33]: def confusion(classes, pred):
             labels = [int(l) for l in list(set(classes))]
             cm = np.zeros((len(labels), len(labels)), float)
             for label in labels:
                 predicted = pred[classes == label]
                 cm[label] = np.bincount(predicted, minlength = len(labels))
             return cm

         def print_confusion_matrix(cm):
             print("\tpredicted")
             print("actual [", cm[0], sep='', end='')
             for i in range(1, len(cm)):
                 print("\n\t", cm[i], sep='',end='')
             print(']')

In [34]: print_confusion_matrix(confusion(test_labels, pred))

        predicted
actual [[10.  3.  1.]
        [ 1. 22.  0.]
        [0. 0. 8.]]
```

The `MclustDA` class can be used in the same manner as the `EDDA` class. `MclustDA` can distinguish more complex classes than `EDDA`, but is compuational more expensive.

```
In [35]: da2 = MclustDA(training, training_labels)

/home/joris/miniconda3/envs/cleannumpy/lib/python3.6/site-packages/mclust/em.py:112: UserWarni
  warnings.warn("singular covariance")
/home/joris/miniconda3/envs/cleannumpy/lib/python3.6/site-packages/mclust/em.py:119: UserWarni
  warnings.warn("mixing proporting fell below threshold")
```

Note that this code migth produce a couple of warnings. The warnings indicate that not all models considered by `MclustDA` can be fitted correctly. This should not be an issue, as the models that do not fit correctly are ignored.

```
In [36]: da2.predict()

Out[36]: array([2, 1, 1, 0, 2, 1, 1, 0, 1, 1, 1, 2, 0, 0, 0, 1, 2, 1, 1, 1, 1, 1,
                2, 1, 0, 2, 1, 2, 0, 1, 0, 1, 1, 2, 1, 2, 0, 1, 1, 1, 1, 1, 0, 1,
                1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 0, 0, 2, 2, 2, 1, 2,
                1, 2, 1, 0, 1, 0, 1, 0, 1, 2, 0, 1, 1, 2, 1, 1, 0, 1, 2, 2, 1, 0,
                0, 1, 2, 1, 1, 2, 1, 1, 2, 2, 2, 1])

In [37]: pred2 = da2.predict(test)
         pred2

Out[37]: array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
                1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 2,
                2])

In [38]: print_confusion_matrix(confusion(test_labels, pred2))

        predicted
actual [[14.  0.  0.]
        [ 1. 22.  0.]
        [0. 0. 8.]]
```

In this case the EDDA and MclustDA classes produce similar results, therefor the EDDA method might be prefered as it takes significantly less compuational resources. In other cases the MclustDA class might perform significantly better than the EDDA class.