



Data Science Initiative

Learning Lab 7: Unsupervised seismic facies classification using Python

Brian Russell

Agenda

- Introduction
- Theory of unsupervised learning using *K*-means and Gaussian Mixture Modeling
- Introduction to the three Python exercises.
- Running the Python code
- Conclusions

Q&A

Guest Host

Brian Russell

Vice President, CGG GeoSoftware
and
Adjunct Professor
Geoscience Department and
CREWES, University of Calgary



Regular Hosts



Marcelo Guarido

Data Scientist with PhD in
Geophysics Head of the
CREWES Data Science
Initiative.



Daniel Trad

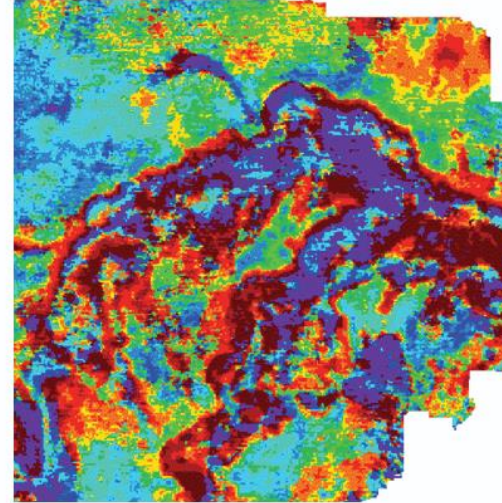
Associate Professor and
Chair in Exploration
Geophysics at University of
Calgary.



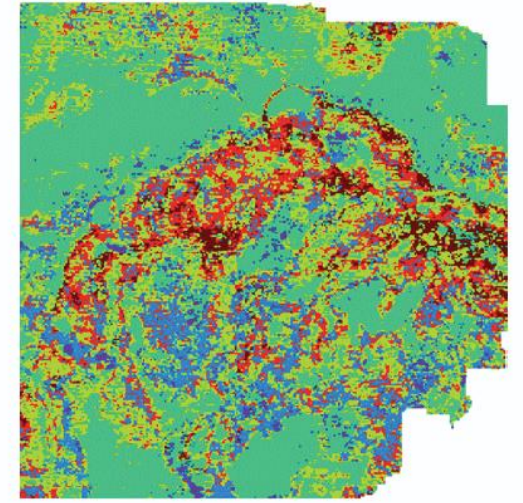
Introduction

- Coléou et al. (2003), in a TLE article, were the first to apply unsupervised clustering techniques to seismic facies classification.
- The left two panels show the Self Organizing Map (SOM) technique with 6 classes (top) and 12 classes (bottom).
- The right two panels show the *K*-means technique with 6 classes (top) and 12 classes (bottom).
- I will not discuss the SOM technique today, but will focus on the *K*-means technique as well as Gaussian Mixture Modelling (GMM), another clustering method.

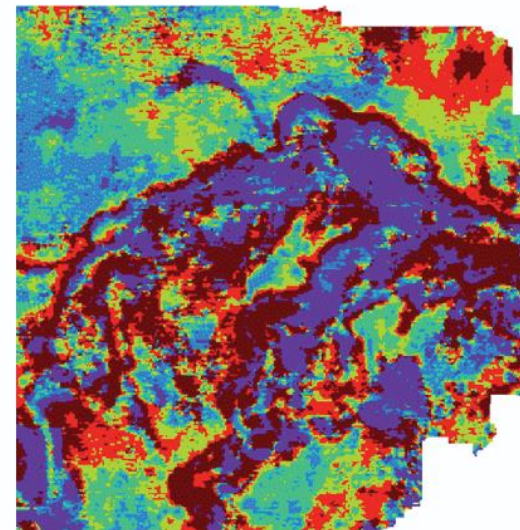
SOM 6 classes



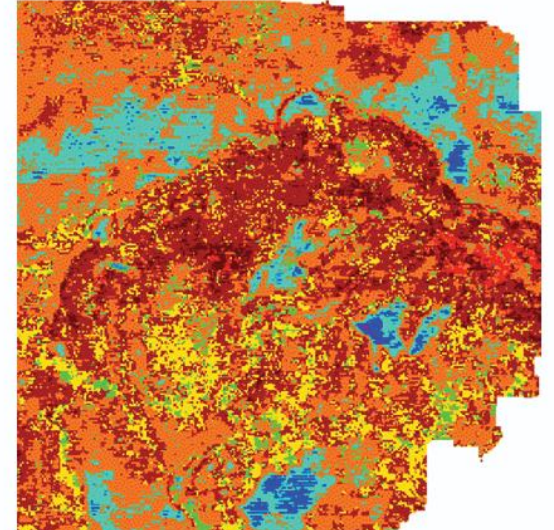
Cluster 6 classes



SOM 12 classes



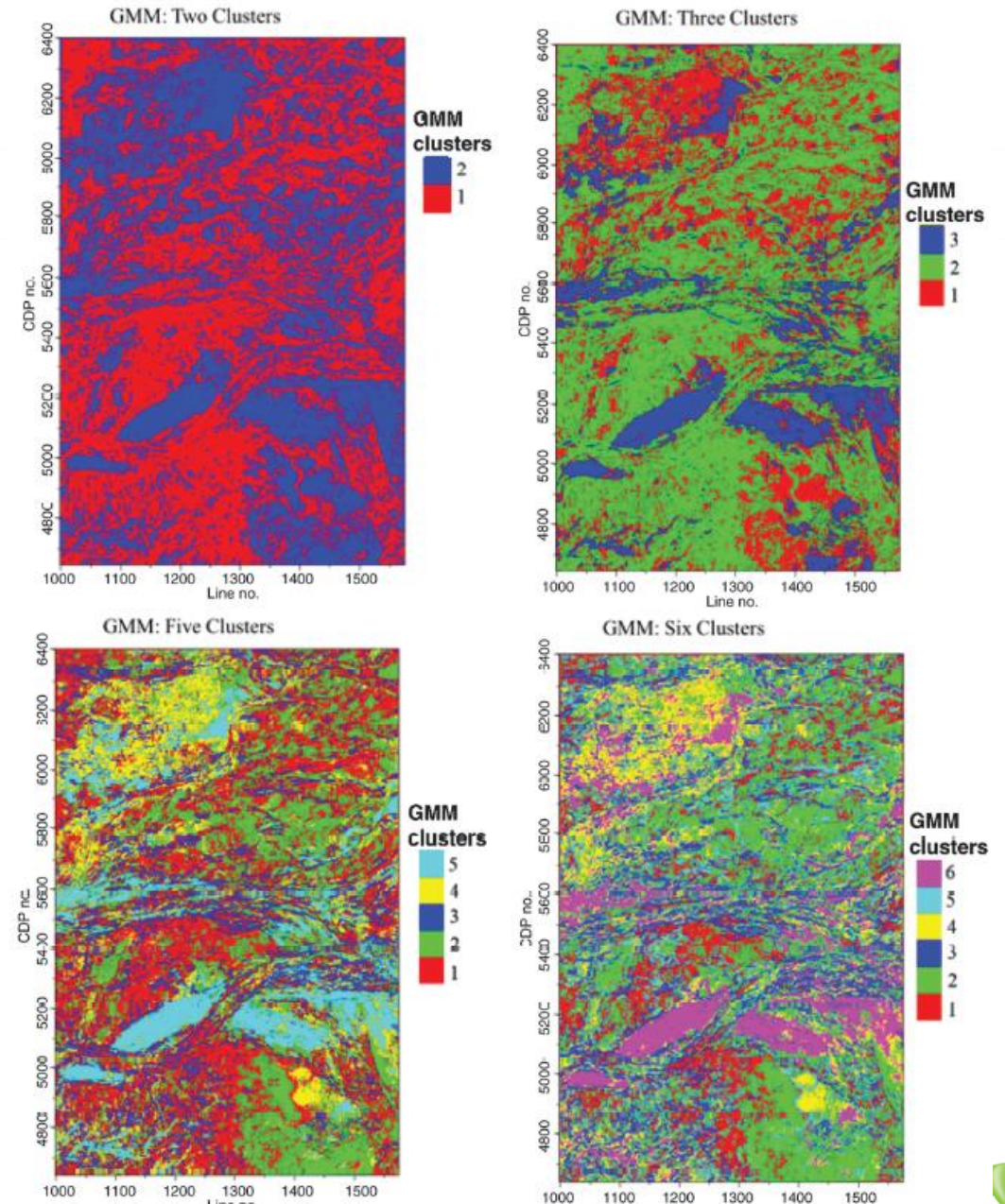
Cluster 12 classes





GMM Example

- Wallet and Hardisty (2019), in an article in Interpretation, applied the GMM technique to seismic facies classification.
- The four panels on the right show the application of GMM to a set of amplitude slices through a seismic volume.
- The upper left shows two clusters, the upper right shows three clusters, the lower left shows five clusters, and the lower right shows six clusters.
- Let's now look at the theory of both K-means and GMM clustering.



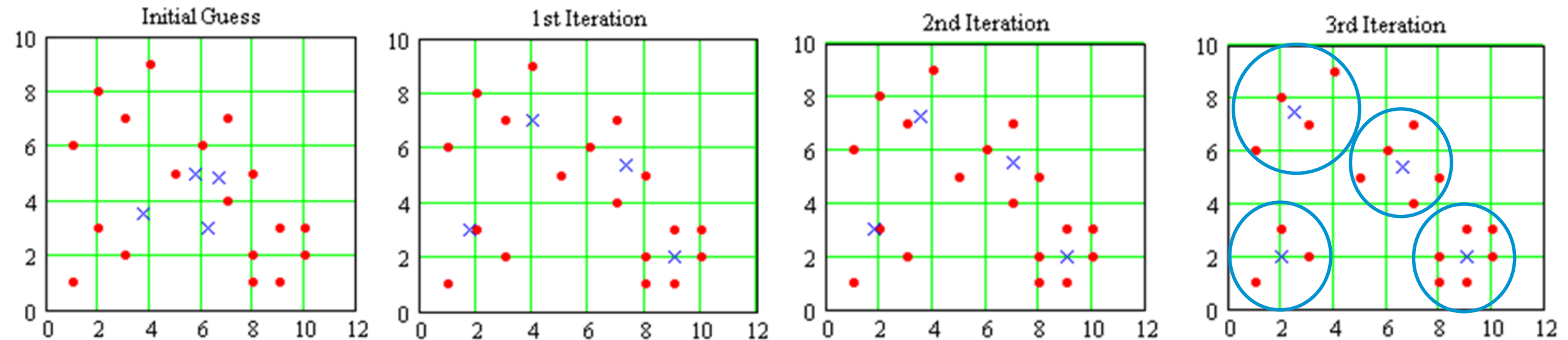


The K -means algorithm

- If we start with N M -dimensional data points (e.g., M attributes) the K -means algorithm divides these points into K clusters.
- The K -means algorithm is implemented as follow:
 - Pick the number of clusters, K , and divide the input data points randomly into these K clusters, each with approximately N/K points.
 - Compute the M -dimensional means of the clusters.
 - Compute the distances between each point and each cluster and assign each point to the cluster for which this distance is a minimum.
 - Re-compute the means based on the new cluster assignments.
 - Iterate through the above three steps until convergence.
- The key assumption is that we know how many clusters are present in the data, so in a typical application you may want to try different values of K .

A simple example of K -means clustering

- To understand K -means, the left figure shows eighteen points that are to be grouped into four clusters, with initial means computed from the data (blue crosses).
- The next three cross-plots shows the updates after the 1st, 2nd and 3rd iterations:

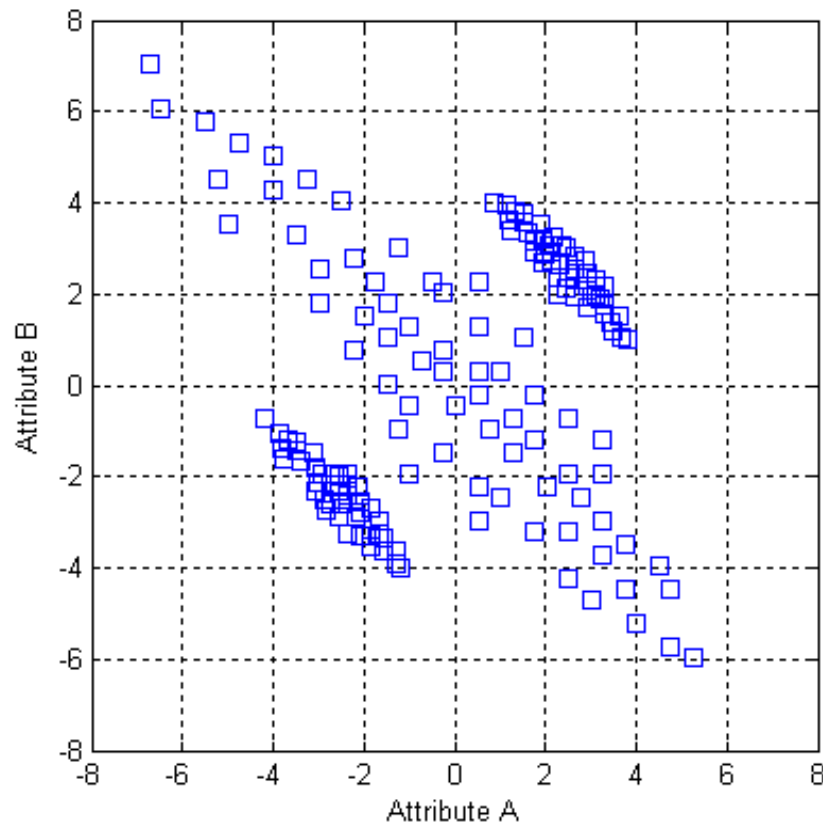


- After the 3rd iteration, the means have “locked in” to the cluster centres, and we can clearly identify the four clusters by drawing circles around the means.

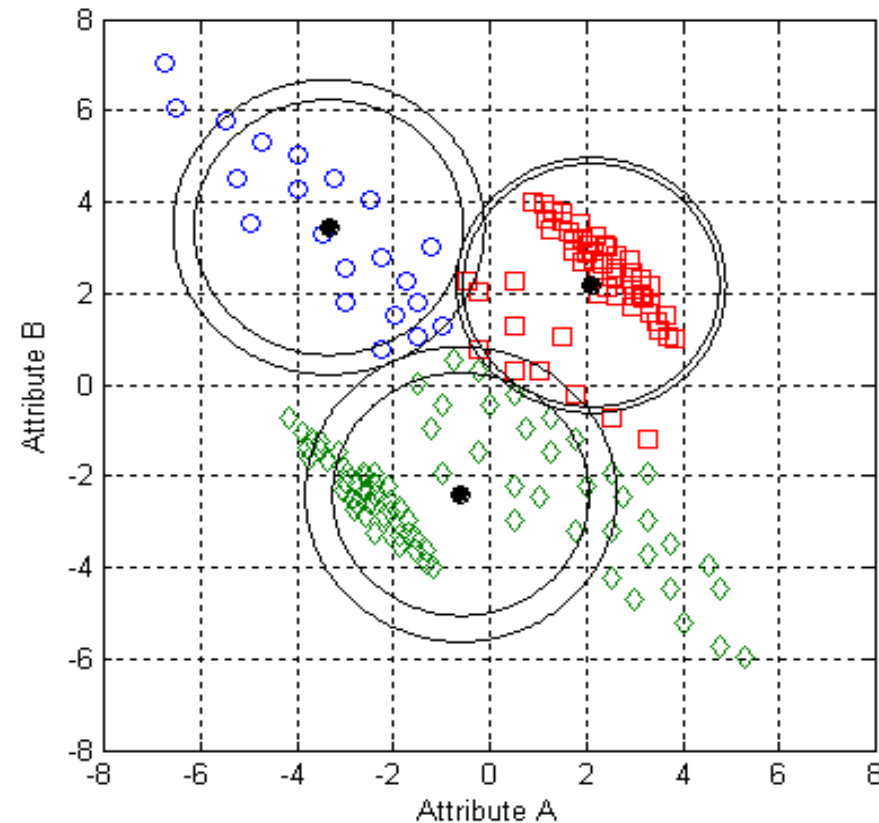


Elliptical clusters

- Here is a dataset with three elliptical clusters, based on an AVO cross-plot:



- Now K -means fails, because it uses Euclidean, not statistical, distance.





The 2-D Gaussian (or Normal) Distribution

- To interpret the elliptical shape of these clusters, we need to understand the Gaussian distribution, which in two dimensions is written:

$$f(\mathbf{x}) = \frac{1}{2\pi|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu})\right], \text{ where}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \boldsymbol{\mu} = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{bmatrix}, \text{ where } \mu_1 \text{ and } \mu_2 \text{ are the two means,}$$

Σ is the covariance matrix with variances σ_1^2 and σ_2^2 , and covariance σ_{12} .

- The term after the -1/2 inside the brackets is the square of the Mahalanobis, or statistical, distance, which is written:

$$\Delta = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu})}$$



The Mahalanobis Distance

- Note that the Mahalanobis, or statistical, distance differs from the Euclidean distance by the inclusion of the inverse covariance matrix, given by:

$$\Sigma^{-1} = \begin{bmatrix} \sigma_1^2 & \sigma_{12} \\ \sigma_{12} & \sigma_2^2 \end{bmatrix}^{-1} = \frac{1}{\sigma_1^2 \sigma_2^2 - \sigma_{12}^2} \begin{bmatrix} \sigma_2^2 & -\sigma_{12} \\ -\sigma_{12} & \sigma_1^2 \end{bmatrix} = \begin{bmatrix} a & -b \\ -b & c \end{bmatrix}$$

- Using zero means and the terms a , b and c , we can expand the square of the Mahalanobis distance as follows:

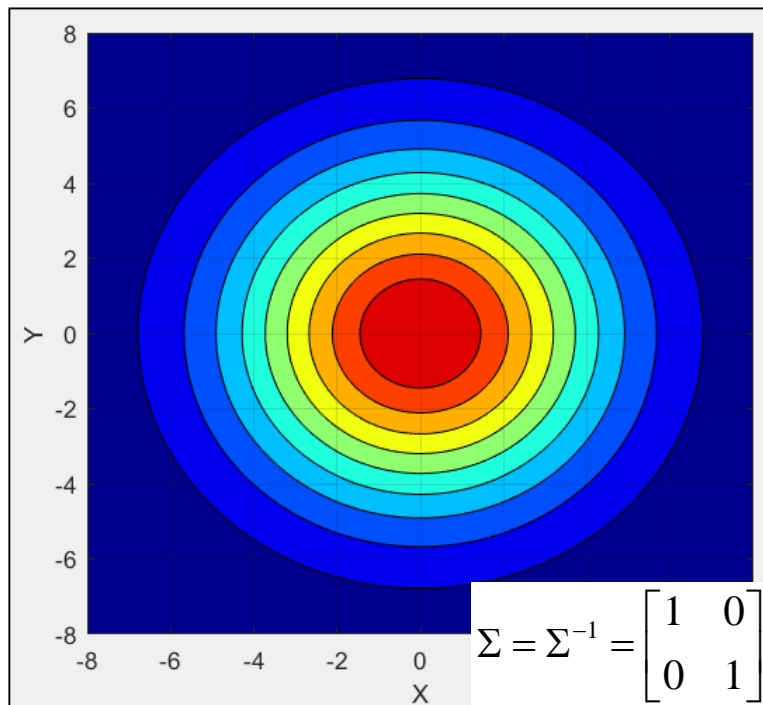
$$\Delta^2 = (x - \mu)^T \Sigma^{-1} (x - \mu) = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} a & -b \\ -b & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = ax_1^2 - 2bx_1x_2 + cx_2^2$$

- Thus, using the Mahalanobis distance squared creates elliptical contours of variance.

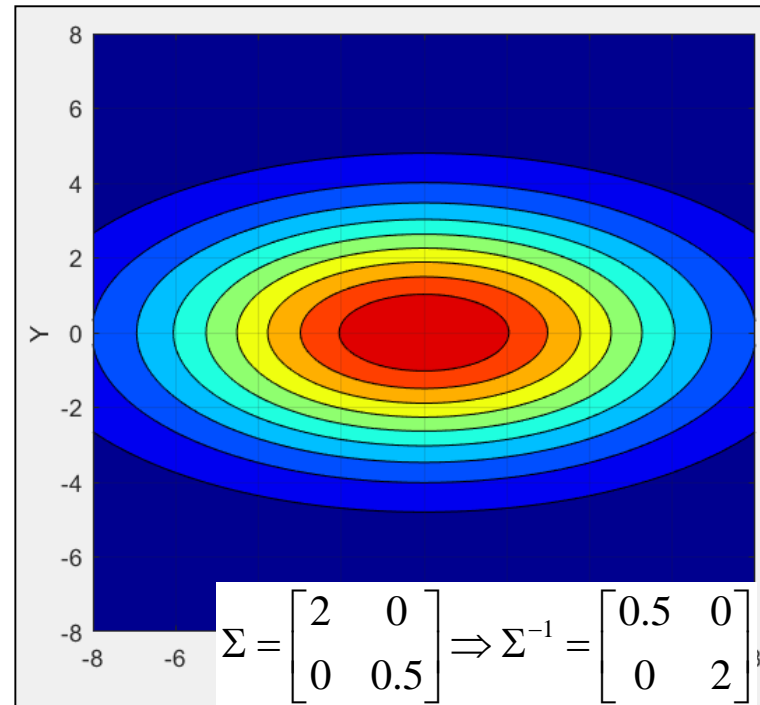
Three special cases

- There are three special cases of the Mahalanobis distance:

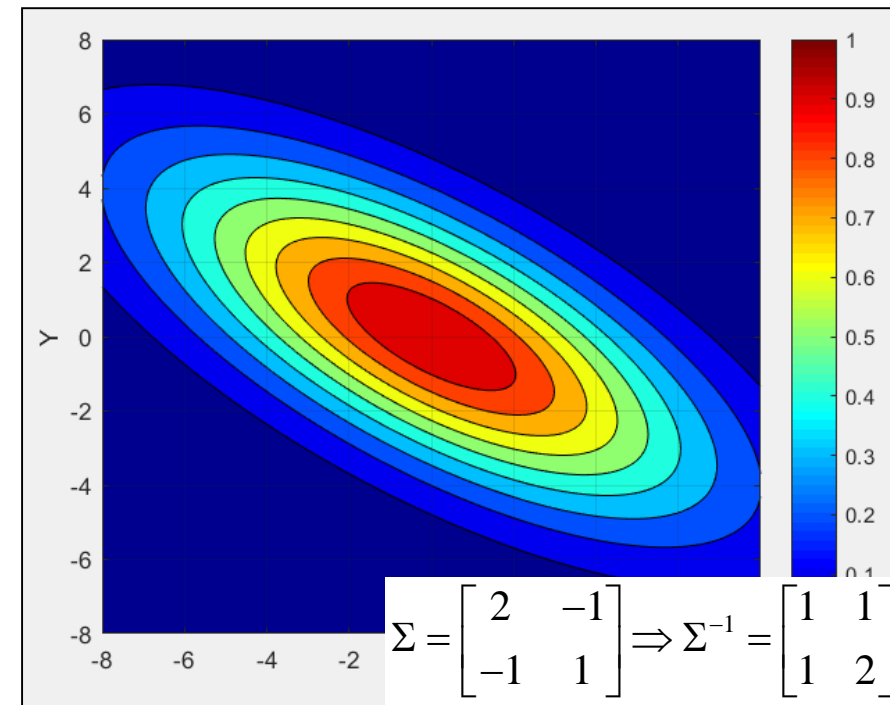
If $b = 0$ and $a = c$, the Mahalanobis distance equals the Euclidean distance:



If $b = 0$ and $a \neq c$, we get vertical ($a < c$) or horizontal ($a > c$) elliptical curves:



If $b \neq 0$, we get tilted ellipses (negative slope if $b < 0$ and vice versa).





The Mahalanobis K -means algorithm

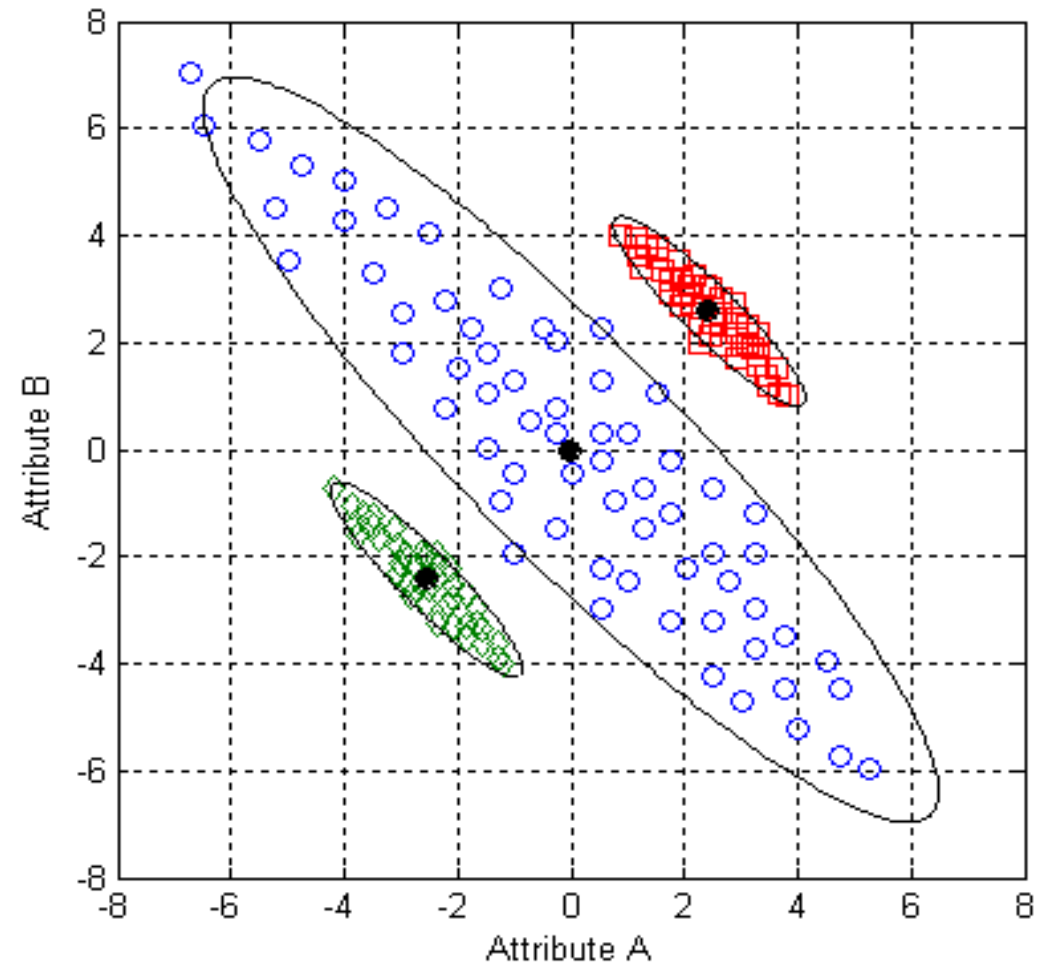
- This suggests a modification of the K -means algorithm as follows:
 - Pick the number of clusters, K , and divide the input data points randomly into these K clusters.
 - Compute the M -dimensional means, μ_k , of the clusters, as well as the covariance matrices Σ_k within each cluster, where $k = 1, 2, \dots, K$.
 - Compute the Mahalanobis distances between each point and cluster and assign each point to the cluster for which this distance is a minimum.
 - Re-compute the means and covariances based on the new cluster assignments.
 - Iterate through the above three steps until convergence.
- Now, let's see how well this algorithm works on our dataset.





Applying the Mahalanobis distance algorithm

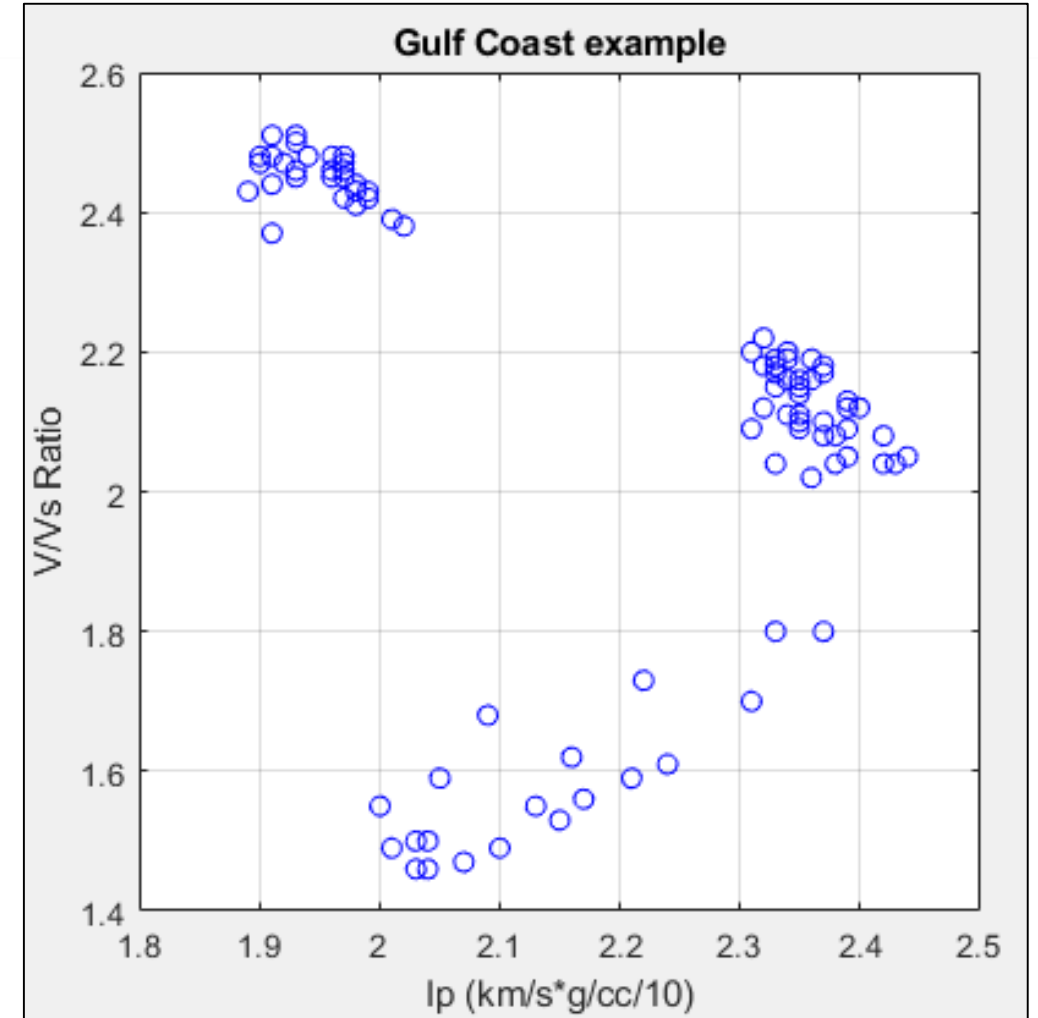
- Here is the result of *K*-means with Mahalanobis distance, and it has worked very well on this dataset.
- Although *K*-means can be modified to search based on statistical distance, the scikit-learn option, which we will be using in the lab today, has not implemented this.
- A more recent clustering method, called Gaussian Mixture Modelling, or GMM, achieves the same result in a different way.
- GMM uses the expectation-maximization (EM) algorithm, which starts with a random guess of the statistics and iterates to a solution.





The Gaussian Mixture Model (GMM)

- To illustrate the Gaussian Mixture Model, or GMM, I will use the example shown on the right.
- Geologically, the upper set of points represent a shale, the lower points a gas sand, and the rightmost points a carbonate.
- The points in this dataset were extracted from the pre-stack inversion of a Gulf Coast dataset.
- Notice that we are cross-plotting inverted V_p/V_s ratio against inverted P-impedance (I_p).
- The points are presented to the algorithm in random order.



The Gaussian Mixture Model (GMM)

- The Gaussian Mixture Model (GMM) is a mixture pdf of N M -dimensional feature vectors \mathbf{x}_n , which are grouped into K classes C_K .
- Each feature vector has a conditional probability given by:

$$p(\mathbf{x}_n | C_k) = \frac{1}{(2\pi)^{M/2} |\Sigma_k|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}_n - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x}_n - \boldsymbol{\mu}_k)\right], \text{ where}$$

$$k = 1, \dots, K, n = 1, \dots, N, \mathbf{x}_n = \begin{bmatrix} x_{1n} \\ \vdots \\ x_{Mn} \end{bmatrix}, \boldsymbol{\mu}_k = \begin{bmatrix} \mu_{1k} \\ \vdots \\ \mu_{Mk} \end{bmatrix} \text{ and } \Sigma_k = \begin{bmatrix} \sigma_{11} & \dots & \sigma_{1M} \\ \vdots & \ddots & \vdots \\ \sigma_{M1} & \dots & \sigma_{MM} \end{bmatrix}.$$

- GMM starts with an initial guess of the means and covariance matrices of each class, and determines the correct values by iterating to a solution.
- Unlike K -means, the data is never physically re-ordered during the process.



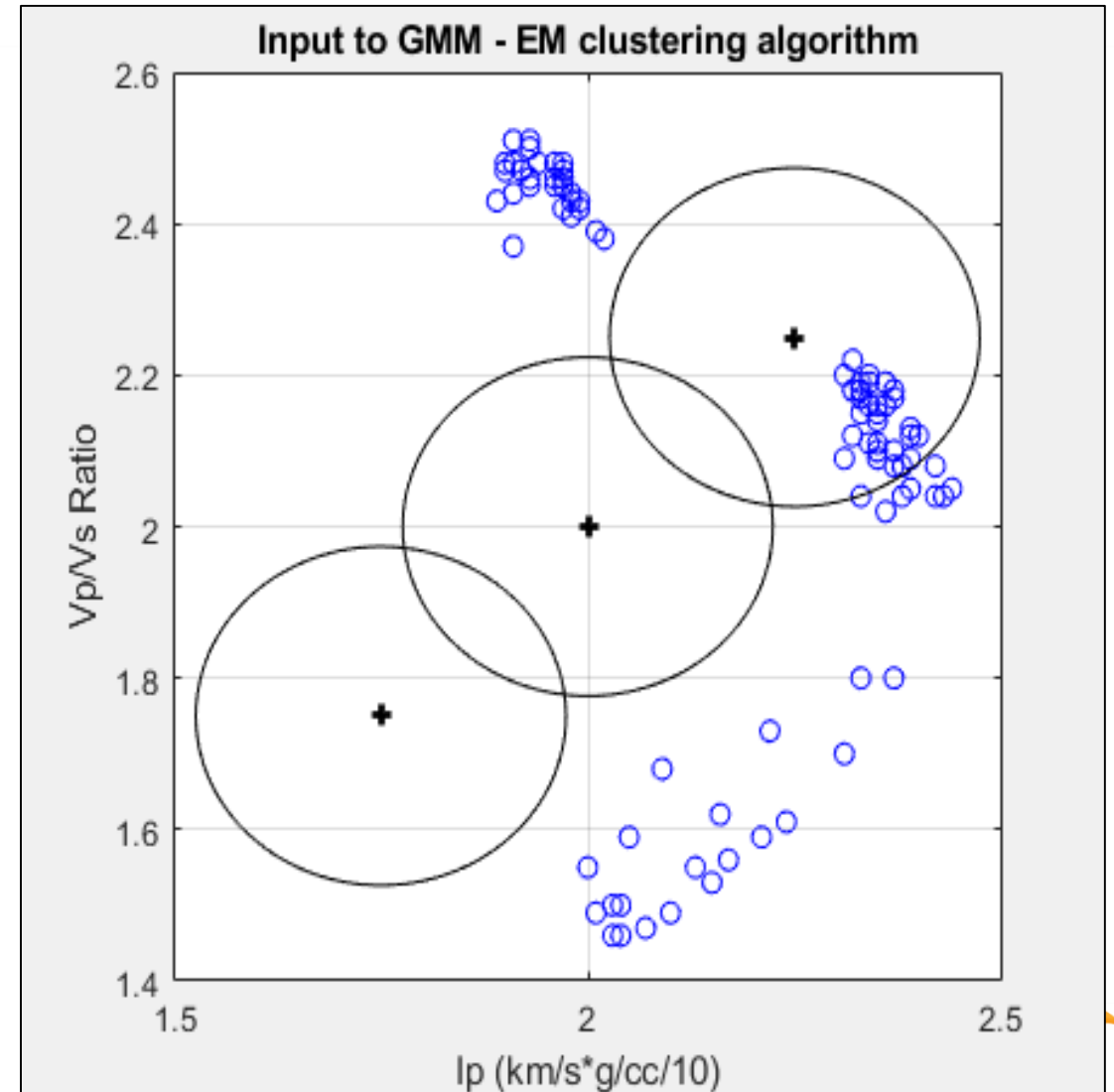


Training the GMM via Expectation-Maximization (*EM*)

- The GMM is trained using the Expectation-Maximization, or *EM*, algorithm.
- The Gaussian functions can have full, diagonal or spherical covariance matrices.
- We will initialize the GMM with three means and covariances given by:

$$\mu_1 = \begin{bmatrix} 2.25 \\ 2.25 \end{bmatrix}, \mu_2 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \mu_3 = \begin{bmatrix} 1.75 \\ 1.75 \end{bmatrix},$$

$$\Sigma_1 = \Sigma_2 = \Sigma_3 = \frac{1}{4} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$





Training the GMM via Expectation-Maximization (*EM*)

- The expectation, or *E*, step involves first determining the “responsibility” for each training point and in each class using Bayes’ Theorem:

$$p(C_k / \mathbf{x}_n) = \frac{p(\mathbf{x}_n | C_k) p(C_k)}{p(\mathbf{x}_n)}, \text{ where } p(\mathbf{x}_n) = \sum_{k=1}^K p(\mathbf{x}_n | C_k) p(C_k).$$

- The maximization, or *M*, step involves re-estimating the component pdfs and mixture weights as follows:

$$\hat{p}(C_k) = \frac{1}{N} \sum_{n=1}^N p(C_k / \mathbf{x}_n), \hat{\boldsymbol{\mu}}_k = \frac{\sum_{n=1}^N p(C_k / \mathbf{x}_n) \mathbf{x}_n}{\sum_{n=1}^N p(C_k / \mathbf{x}_n)}, \text{ and } \hat{\boldsymbol{\Sigma}}_k = \frac{\sum_{n=1}^N p(C_k / \mathbf{x}_n) (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)^T (\mathbf{x}_n - \hat{\boldsymbol{\mu}}_k)}{\sum_{n=1}^N p(C_k / \mathbf{x}_n)}.$$

- This step is repeated until convergence.



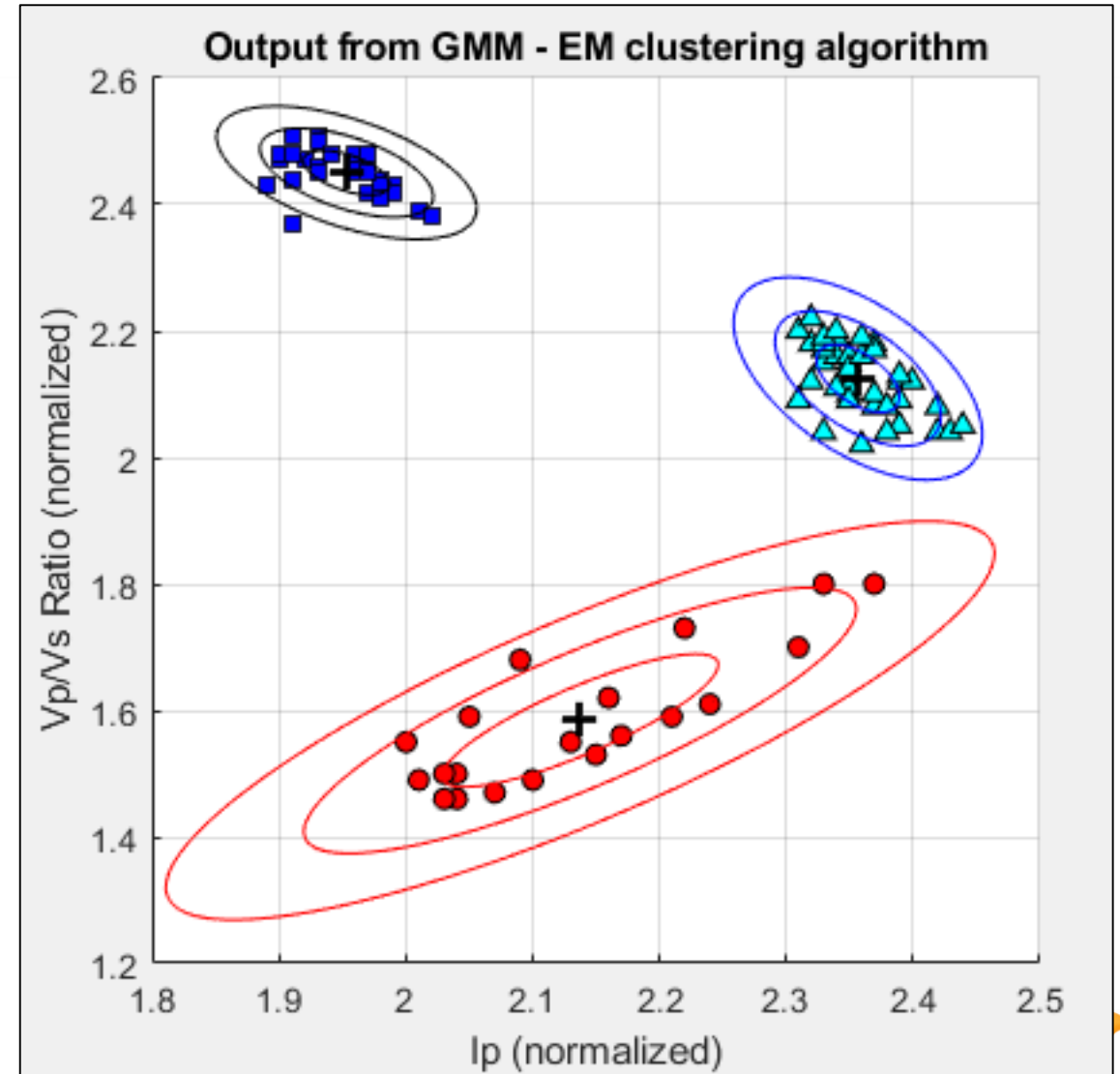
Clustered result

- This figure shows our clustered result, with their Gaussian contours shown.
- The colours can be thought of as “labels”, which will tell us how to classify the points.
- The final statistics are:

$$\mu_1 = \begin{bmatrix} 2.1375 \\ 1.5840 \end{bmatrix}, \Sigma_1 = \begin{bmatrix} 0.0119 & 0.0096 \\ 0.0096 & 0.0110 \end{bmatrix}.$$

$$\mu_2 = \begin{bmatrix} 1.9529 \\ 2.4503 \end{bmatrix}, \Sigma_2 = \begin{bmatrix} 0.0012 & -0.0006 \\ -0.0006 & 0.0012 \end{bmatrix}.$$

$$\mu_3 = \begin{bmatrix} 2.3571 \\ 2.1248 \end{bmatrix}, \Sigma_3 = \begin{bmatrix} 0.0011 & -0.001 \\ -0.001 & 0.0029 \end{bmatrix}.$$





The three Python examples

- Next, I will illustrate how to implement K -means and GMM in Python with scikit-learn.
- I will use the following three examples:
 - The classic “blobs” dataset, which is an 2D example found in every Python machine learning book using a built-in scikit-learn option (all the authors shamelessly copy their examples from the scikit-learn tutorials!) .
 - A more realistic set of three elliptical 2D clusters.
 - A multi-dimensional seismic example that comes from the Blackfoot dataset and involves performing facies classification.
- Before looking at the Python code, let me explain each of the examples in a little more detail.

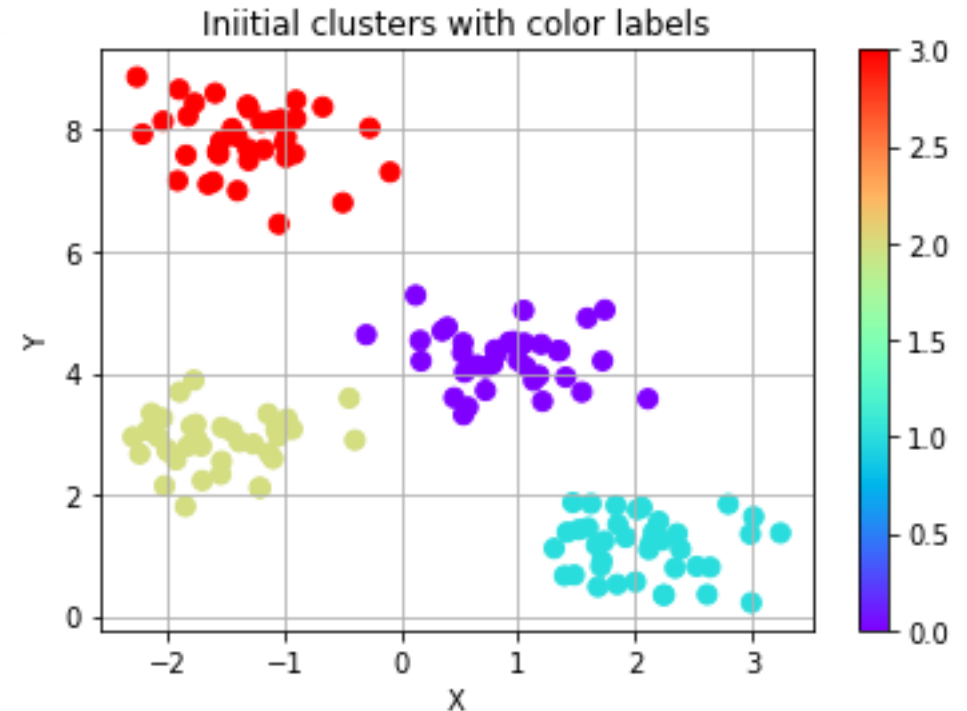




Example 1: Making “blobs”

- The classic “blobs” dataset is a 2D example found in every Python machine learning book using a built-in scikit-learn option.
- It is initialized as follows, producing the result shown on the right:

```
7 # Example taken from Data Science Handbook
8 from sklearn.datasets import make_blobs
9 X,y=make_blobs(n_samples=150,n_features=2,centers=4,
10               cluster_std=0.5,shuffle=True,random_state=0)
```



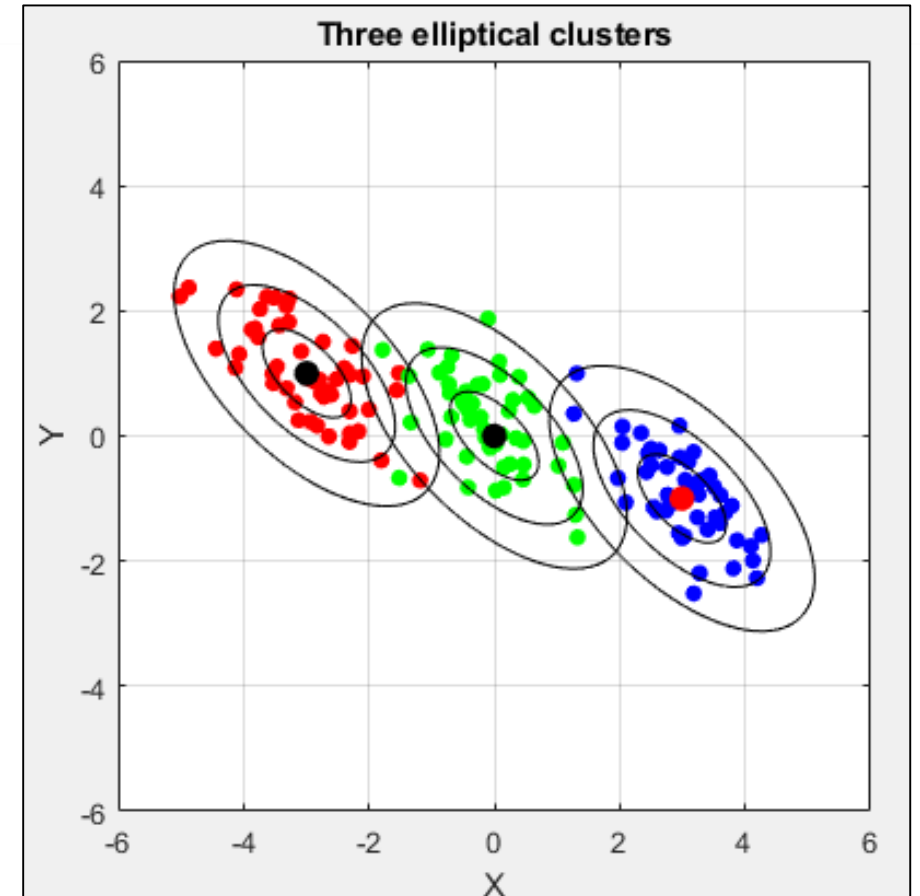
- Note that I have used the “make_blobs” option to create four clusters of points with a total number of 150 samples, each with a standard deviation of 0.5.
- Since there is no covariance term, the blobs will all be circular.



Example 2: Three elliptical clusters

- To create clusters with elliptical shapes I used mvnrnd (multivariate random) in Matlab and then randomized their order.
- Each cluster had 50 points and their final form is shown in the plot on the right.
- The three means are different but the covariance matrix is the same for all three:

$$\mu_1 = \begin{bmatrix} -3 \\ 1 \end{bmatrix}, \mu_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mu_3 = \begin{bmatrix} 3 \\ -1 \end{bmatrix},$$
$$\text{and } \Sigma = \begin{bmatrix} 0.5 & -0.3 \\ -0.3 & 0.5 \end{bmatrix}.$$

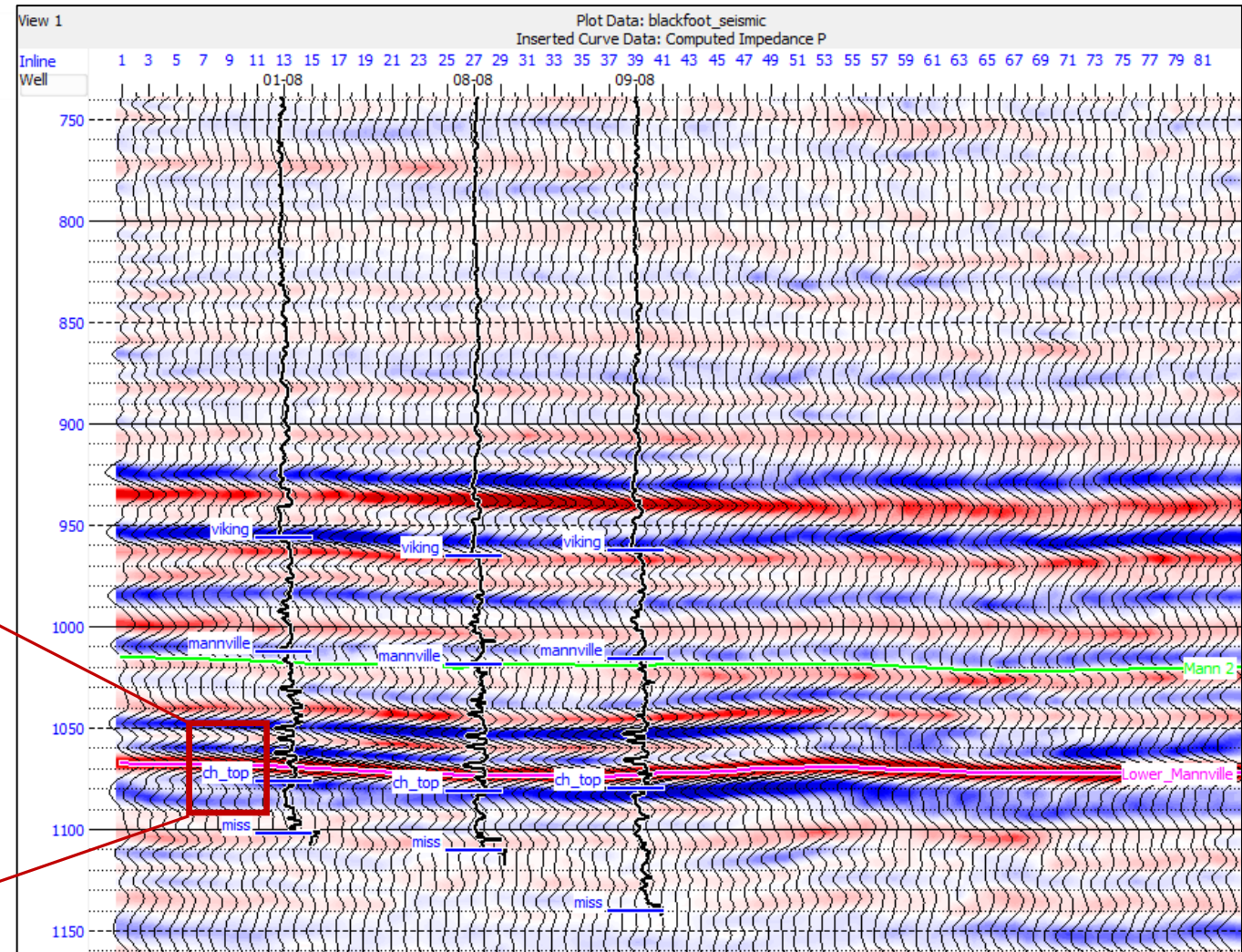
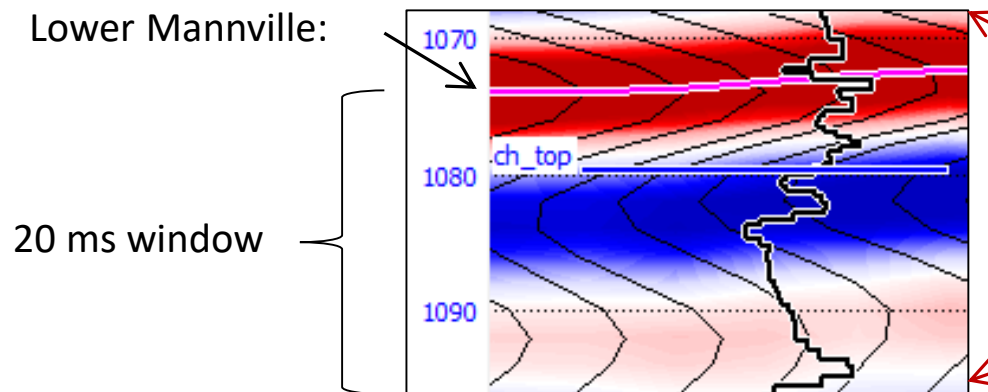


- I saved the X and Y coordinates of the points and their labels in the separate files: Three_Clusters.txt and Three_Clusters_Labels, to be read into Python.



Example 3: The Blackfoot dataset

- In our third example, we will apply the *K*-means and GMM algorithms to the Blackfoot dataset from Alberta, shown on the right.
- The algorithms will be applied to a 10 ms (5 sample) window below the Lower Mannville event.

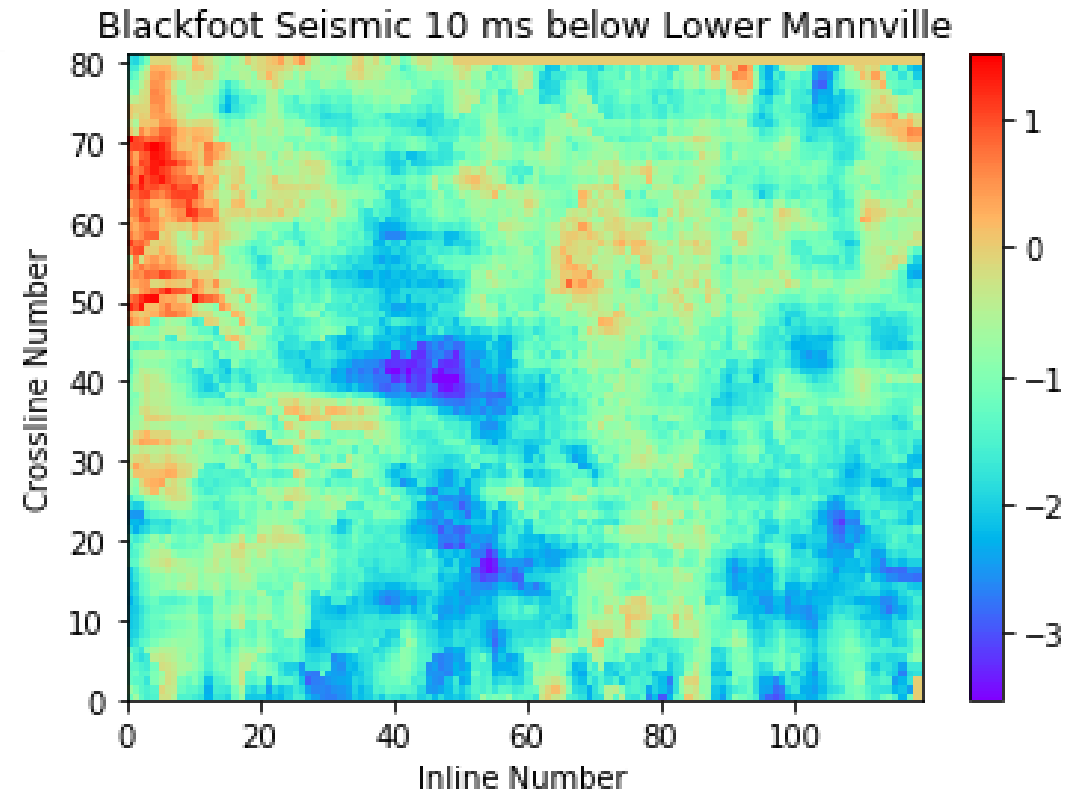


- This means that we are doing the clustering in 5-dimensional space.



Example 3: The Blackfoot dataset

- The Blackfoot dataset therefore consists of 5 data slices from the seismic volume between times of 10 ms and 18 ms below the Lower Manville.
- I extracted the data slices the HampsonRussell GeoView program using our Python Ecosystem.
- I then saved it as a 5 column ASCII file called Blackfoot_data.txt, which will be read into the Python program.
- The figure shown here is one of the data slices after being re-shaped to its correct map coordinates, plotted in Python.





References

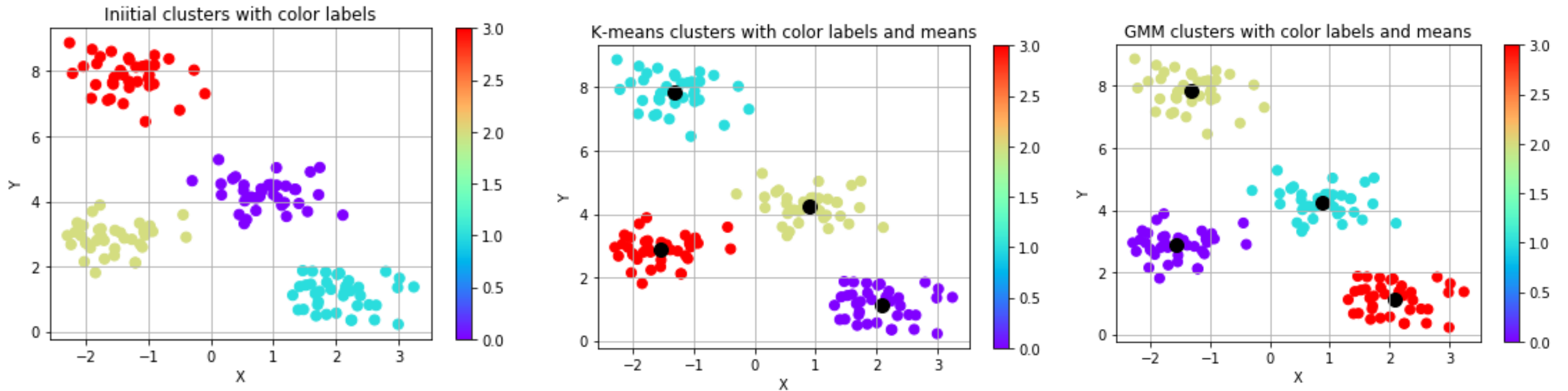
Coléou, T., Poupon, M., and Azbel, K., 2003, Unsupervised seismic facies classification: A review and comparison of techniques and implementation: The Leading Edge, 942-953.

Wallet, B.C. and Hardisty, R., 2019, Unsupervised seismic facies using Gaussian mixture models: Interpretation, August, 2019, SE93 – SE111.





Appendix 1: Results

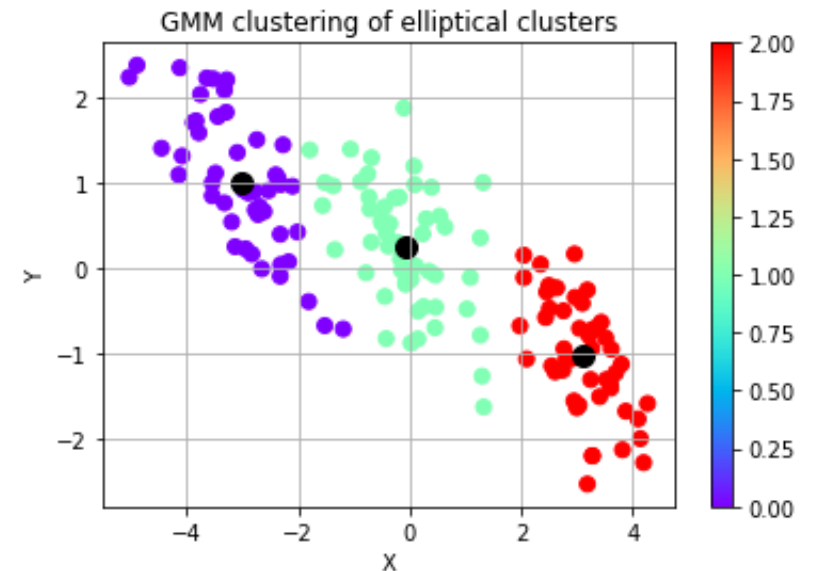
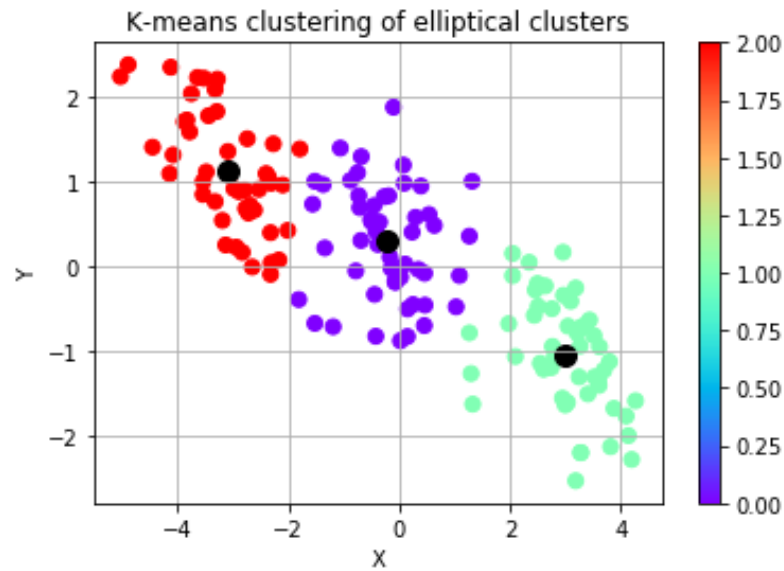
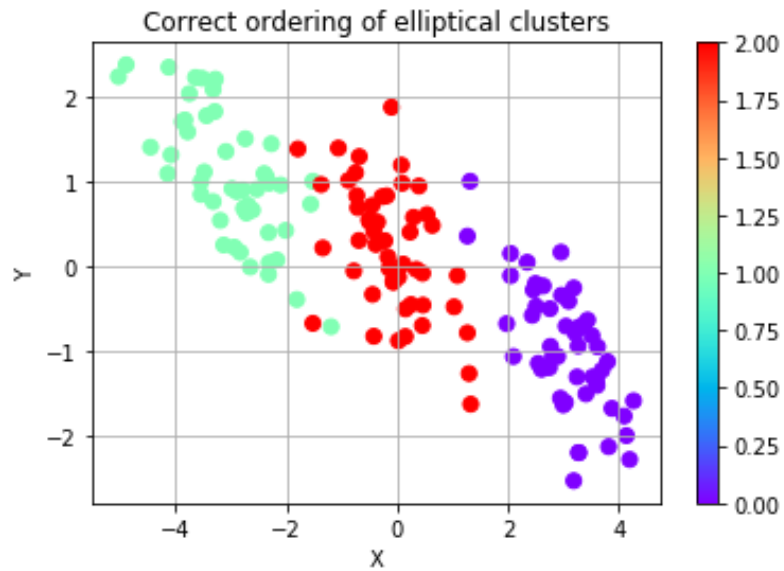


Blobs example, with std. dev. = 0.5





Appendix 1: Results

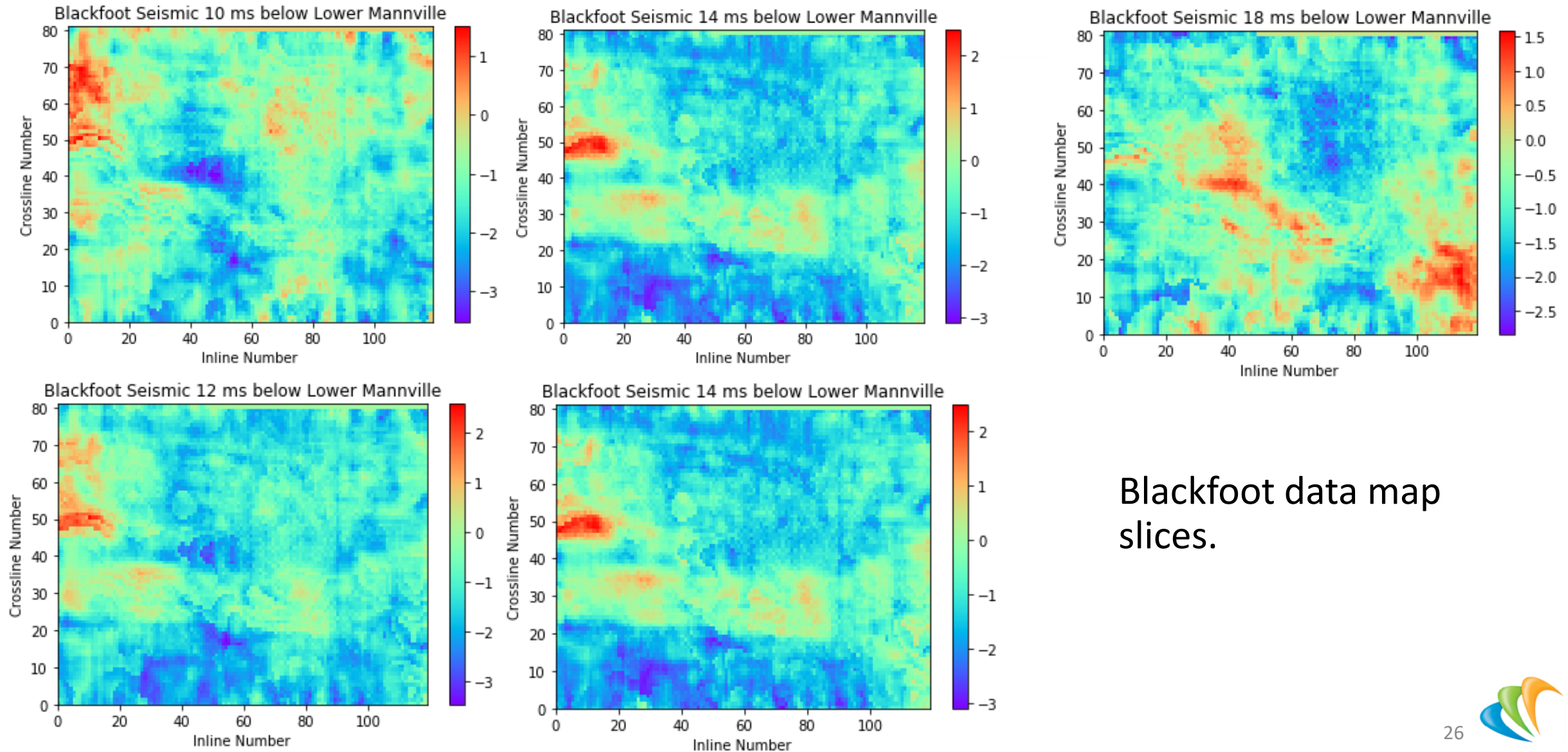


Elliptical clusters example.





Appendix 1: Results

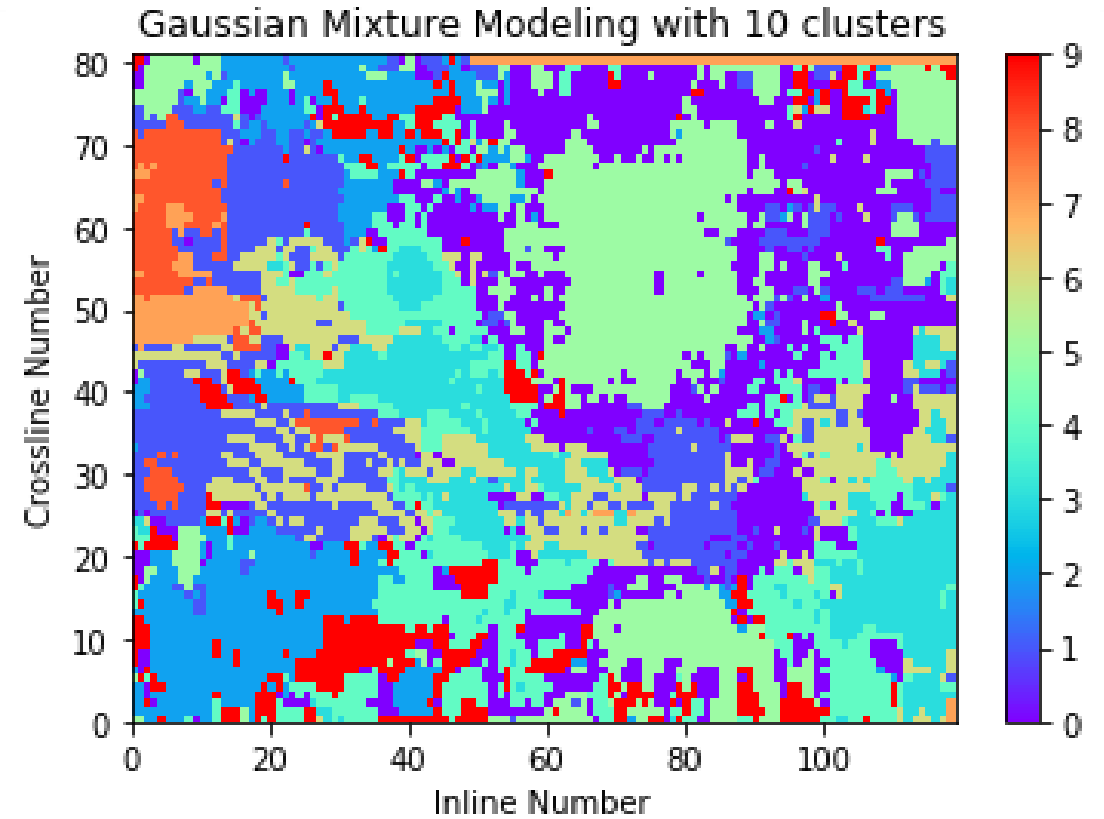
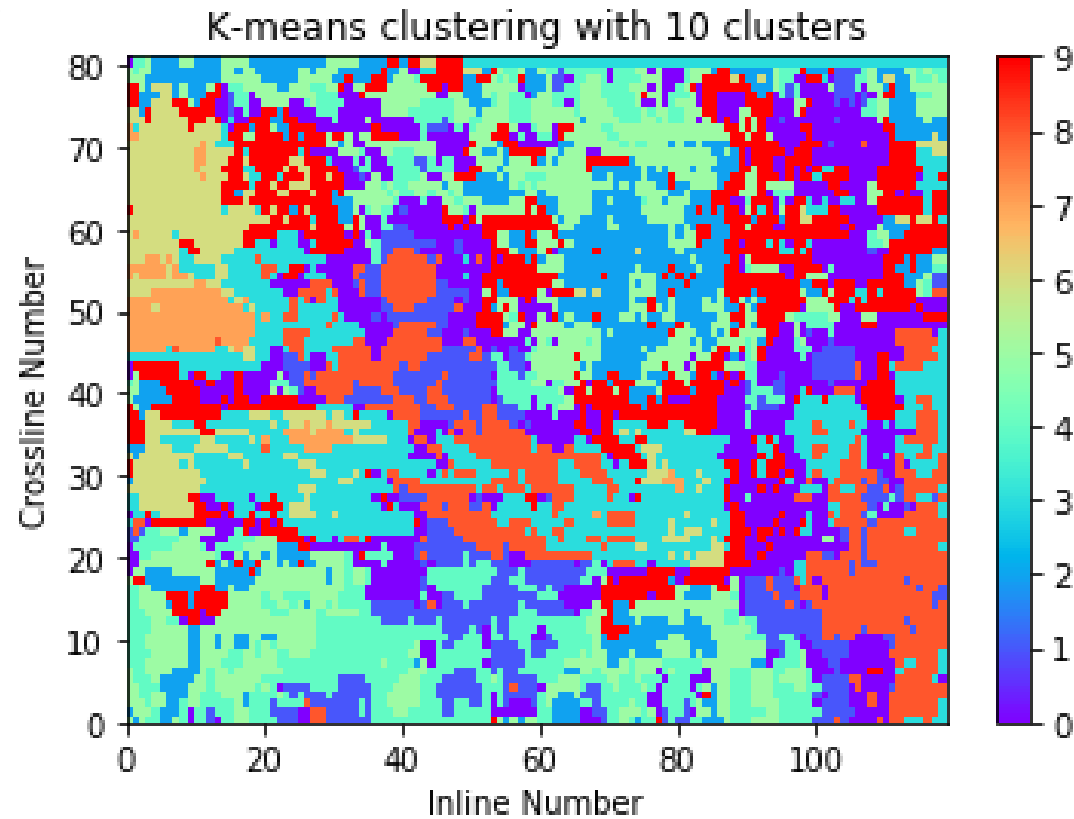


Blackfoot data map slices.





Appendix 1: Results



Blackfoot data clustering results with 10 clusters.





Appendix 2: scikit-learn documentation

In this appendix, I have included the documentation for the options used in Python from scikit-learn.





make_blobs in scikit-learn:

sklearn.datasets.make_blobs

Parameters:

n_samples : *int or array-like, optional (default=100)*

If int, it is the total number of points equally divided among clusters. If array-like, each element of the sequence indicates the number of samples per cluster.

Changed in version v0.20: one can now pass an array-like to the `n_samples` parameter

n_features : *int, optional (default=2)*

The number of features for each sample.

centers : *int or array of shape [n_centers, n_features], optional*

(default=None) The number of centers to generate, or the fixed center locations. If `n_samples` is an int and `centers` is None, 3 centers are generated. If `n_samples` is array-like, `centers` must be either None or an array of length equal to the length of `n_samples`.

cluster_std : *float or sequence of floats, optional (default=1.0)*

The standard deviation of the clusters.

center_box : *pair of floats (min, max), optional (default=(-10.0, 10.0))*

The bounding box for each cluster center when centers are generated at random.

shuffle : *boolean, optional (default=True)*

Shuffle the samples.

random_state : *int, RandomState instance, default=None*

Determines random number generation for dataset creation. Pass an int for reproducible output across multiple function calls. See [Glossary](#).

return_centers : *bool, optional (default=False)*

If True, then return the centers of each cluster





make-blobs in scikit-learn:

Returns:

X : array of shape [n_samples, n_features]

The generated samples.

y : array of shape [n_samples]

The integer labels for cluster membership of each sample.

centers : array, shape [n_centers, n_features]

The centers of each cluster. Only returned if `return_centers=True`.





K-means in scikit-learn:

sklearn.cluster.KMeans

```
class sklearn.cluster.KMeans(n_clusters=8, *, init='k-means++', n_init=10, max_iter=300, tol=0.0001,
precompute_distances='deprecated', verbose=0, random_state=None, copy_x=True, n_jobs='deprecated', algorithm='auto') \[source\]
```

Parameters:

n_clusters : int, default=8

The number of clusters to form as well as the number of centroids to generate.

init : {'k-means++', 'random', ndarray, callable}, default='k-means++'

Method for initialization:

'k-means++': selects initial cluster centers for k-mean clustering in a smart way to speed up convergence. See section Notes in `k_init` for more details.

'random': choose `n_clusters` observations (rows) at random from data for the initial centroids.

If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.

If a callable is passed, it should take arguments X, n_clusters and a random state and return an initialization.

n_init : int, default=10

Number of time the k-means algorithm will be run with different centroid seeds. The final results will be the best output of n_init consecutive runs in terms of inertia.

max_iter : int, default=300

Maximum number of iterations of the k-means algorithm for a single run.

tol : float, default=1e-4

Relative tolerance with regards to Frobenius norm of the difference in the cluster centers of two consecutive iterations to declare convergence.





K-means in scikit-learn:

precompute_distances : {'auto', True, False}, default='auto'

Precompute distances (faster but takes more memory).

'auto' : do not precompute distances if $n_samples * n_clusters > 12$ million. This corresponds to about 100MB overhead per job using double precision.

True : always precompute distances.

False : never precompute distances.

Deprecated since version 0.23: 'precompute_distances' was deprecated in version 0.22 and will be removed in 0.25. It has no effect.

verbose : int, default=0

Verbosity mode.

random_state : int, RandomState instance, default=None

Determines random number generation for centroid initialization. Use an int to make the randomness deterministic. See [Glossary](#).

copy_x : bool, default=True

When pre-computing distances it is more numerically accurate to center the data first. If copy_x is True (default), then the original data is not modified. If False, the original data is modified, and put back before the function returns, but small numerical differences may be introduced by subtracting and then adding the data mean. Note that if the original data is not C-contiguous, a copy will be made even if copy_x is False. If the original data is sparse, but not in CSR format, a copy will be made even if copy_x is False.

n_jobs : int, default=None

The number of OpenMP threads to use for the computation. Parallelism is sample-wise on the main cython loop which assigns each sample to its closest center.

None or -1 means using all processors.





K-means in scikit-learn:

algorithm : {"auto", "full", "elkan"}, default="auto"

K-means algorithm to use. The classical EM-style algorithm is "full". The "elkan" variation is more efficient on data with well-defined clusters, by using the triangle inequality. However it's more memory intensive due to the allocation of an extra array of shape (n_samples, n_clusters).

For now "auto" (kept for backward compatibility) chooses "elkan" but it might change in the future for a better heuristic.

Changed in version 0.18: Added Elkan algorithm

Attributes:

cluster_centers_ : ndarray of shape (n_clusters, n_features)

Coordinates of cluster centers. If the algorithm stops before fully converging (see `tol` and `max_iter`), these will not be consistent with `labels_`.

labels_ : ndarray of shape (n_samples,)

Labels of each point

inertia_ : float

Sum of squared distances of samples to their closest cluster center.

n_iter_ : int

Number of iterations run.





GMM in scikit-learn:

sklearn.mixture.GaussianMixture

```
class sklearn.mixture.GaussianMixture(n_components=1, *, covariance_type='full', tol=0.001, reg_covar=1e-06, max_iter=100,
n_init=1, init_params='kmeans', weights_init=None, means_init=None, precisions_init=None, random_state=None,
warm_start=False, verbose=0, verbose_interval=10)
```

[source]

Parameters:

n_components : *int, defaults to 1.*

The number of mixture components.

covariance_type : {'full' (default), 'tied', 'diag', 'spherical'}

String describing the type of covariance parameters to use. Must be one of:

'full'

each component has its own general covariance matrix

'tied'

all components share the same general covariance matrix

'diag'

each component has its own diagonal covariance matrix

'spherical'

each component has its own single variance





GMM in scikit-learn:

tol : float, defaults to 1e-3.

The convergence threshold. EM iterations will stop when the lower bound average gain is below this threshold.

reg_covar : float, defaults to 1e-6.

Non-negative regularization added to the diagonal of covariance. Allows to assure that the covariance matrices are all positive.

max_iter : int, defaults to 100.

The number of EM iterations to perform.

n_init : int, defaults to 1.

The number of initializations to perform. The best results are kept.

init_params : {'kmeans', 'random'}, defaults to 'kmeans'.

The method used to initialize the weights, the means and the precisions. Must be one of:

```
'kmeans' : responsibilities are initialized using kmeans.  
'random' : responsibilities are initialized randomly.
```

weights_init : array-like, shape (n_components,), optional

The user-provided initial weights, defaults to None. If it None, weights are initialized using the `init_params` method.

means_init : array-like, shape (n_components, n_features), optional

The user-provided initial means, defaults to None, If it None, means are initialized using the `init_params` method.





GMM in scikit-learn:

Attributes:

weights_ : array-like, shape (n_components,)

The weights of each mixture components.

means_ : array-like, shape (n_components, n_features)

The mean of each mixture component.

covariances_ : array-like

The covariance of each mixture component. The shape depends on `covariance_type`:

```
(n_components,)           if 'spherical',  
(n_features, n_features)  if 'tied',  
(n_components, n_features) if 'diag',  
(n_components, n_features, n_features) if 'full'
```

