Chapter 18: Clustering and classification

The method of determining the properties of the thematic classes directly from the reference data is called **supervised classification**, because the analyst actually “supervises” how the discriminant functions of the classes are formed by providing the reference data.

Unsupervised classification methods on the other hand group data points (e.g. pixels) together based on their similarities, with no information from the user about which ones belong together. The user selects the independent or predictor variables of interest, and the chosen algorithm does the rest. This doesn’t mean that you don’t need to know what you’re classifying, however. Once a classification is produced, it’s up to the user to interpret it and decide which types of features correspond to which generated classes, or if they even do correspond nicely.

Unsupervised classification is also called **clustering**.

K-Means clustering algorithm

The *K-Means* method is one of the most common unsupervised classification approach.

The algorithm requires an arbitrarily specified initial cluster centres that are represented by the means of the data points assigned to them. As a naïve solution, the user only defines the number of clusters and random data points are selected as their initial centers.

This will generate a very crude set of clusters. The data points are then reassigned to the cluster with the closest center, and the centers are recomputed. The process is repeated as many times as necessary such that there is no further movement of the data points between clusters. In practice, with large data sets, the process is not run to completion and some other stopping rule is used.

Considering the squared distance between each data point and the respective cluster center as the *squared error*, the sum of squared errors (*SSE*) progressively reduces with each iteration. If the *Euclidean distance* is used as a metric, this simply means to accumulate the squared distances for all points and their respective cluster center.

Although no general proof of convergence exists for this algorithm, it can be expected to yield acceptable results when the data exhibit characteristic pockets which are relatively far from each other. In most practical cases the application of this algorithm will require experimenting with various values of inital clusters (the value of k), as well as different choices of starting configurations.

Clustering raster data

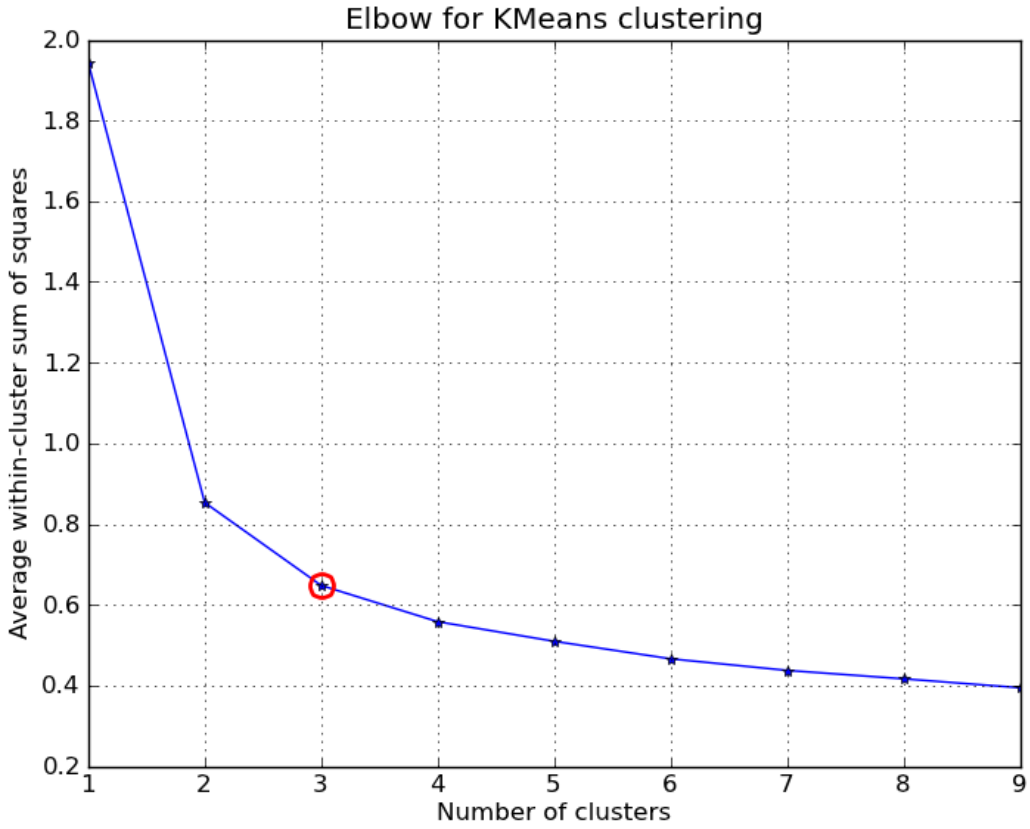
The K-Means method can be use with an arbitrary *distance function*. For raster imagery the distance is computed as if the pixel values were coordinates. For example, if the insensity values of two RGB pixels were $(25, 42, 37)$ and $(31, 40, 32)$, the squared distance would be $(25-31)^2 + (42-40)^2 + (37-32)^2 = 65$ in the 3 dimensional spectral space, no matter where the pixels were in relation to each other spatially.

Elbow method

One of the most challenging tasks in the K-Means clustering algorithm is to choose the right value of the clusters (the value of k). What should be the right value of k and how to choose it?

The *Elbow Method* is one of the most popular methods to determine the optimal value of k. The idea is to run K-Means clustering on the dataset for a range of values of k (e.g. from 1 to 10), and for each value of k calculate the sum of squared errors (*SSE*).

Then, visualize a line chart of the SSE for each value of k. If the line chart looks like an arm, then the "elbow" on the arm is the value of k that is the best. The idea is that we want a small SSE, but that the SSE tends to decrease toward 0 as we increase k. (The SSE is 0 when k is equal to the number of data points in the dataset, because then each data point has its own cluster, and there is no error between it and the center of its cluster.) Hence we select the value of k at the “elbow”, i.e. the point after which the line chart starts decreasing in a linear fashion.



K-Means clustering in Python

Scikit-learn (also known as *sklearn*) is a machine learning library for Python. It features various classification, regression and clustering algorithms including *k-means*.

How to install *scikit-learn*?

If you have Anaconda installed, then `scikit-learn` was already installed together with it.

If you have a standalone Python3 and Jupyter Notebook installation, open a command prompt / terminal and type in:

```
pip3 install scikit-learn
```

Clustering vector data

Read the `hungary_cities.shp` shapefile located in the `data` folder. This dataset contains both scalar and spatial data of the Hungarian cities, and should be familiar from Chapter 15 (15_graph_spanning_tree.pdf).

In [1]:

```
import geopandas as gpd
from sklearn.cluster import KMeans

cities_gdf = gpd.read_file('../data/hungary_cities.shp')
display(cities_gdf)
```

	Id	County	City	Status	KSH	geometry
0	1	FEJÉR	Aba	town	17376	POINT (610046.800 187639.000)
1	2	BARANYA	Abaliget	town	12548	POINT (577946.100 89280.800)
2	3	HEVES	Abasár	town	24554	POINT (721963.700 273880.300)
3	4	BORSOD-ABAUJ-ZEMPLÉN	Abaújalpár	town	15662	POINT (812129.200 331508.200)
4	5	BORSOD-ABAUJ-ZEMPLÉN	Abaújkér	town	26718	POINT (809795.600 331138.300)
...	...	...	...	...	...	...
3142	3143	GYÕR-MOSON-SOPRON	Zsira	town	04622	POINT (471324.200 237577.200)
3143	3144	CSONGRÁD	Zsombó	town	17765	POINT (721098.100 109690.000)
3144	3145	BORSOD-ABAUJ-ZEMPLÉN	Zsujta	town	11022	POINT (815027.400 353143.100)
3145	3146	SZABOLCS-SZATMÁR-BEREG	Zsurk	town	13037	POINT (884847.700 344952.800)
3146	3147	BORSOD-ABAUJ-ZEMPLÉN	Zubogy	town	19105	POINT (763123.300 338338.600)

3147 rows × 6 columns

Fetch points for cities:

In [2]:

```
points = [(geom.x, geom.y) for geom in cities_gdf.geometry]
print("Number of points: {0}".format(len(points)))
```

```
Number of points: 3147
```

Cluster the points using the *K-Means algorithm*:

In [3]:

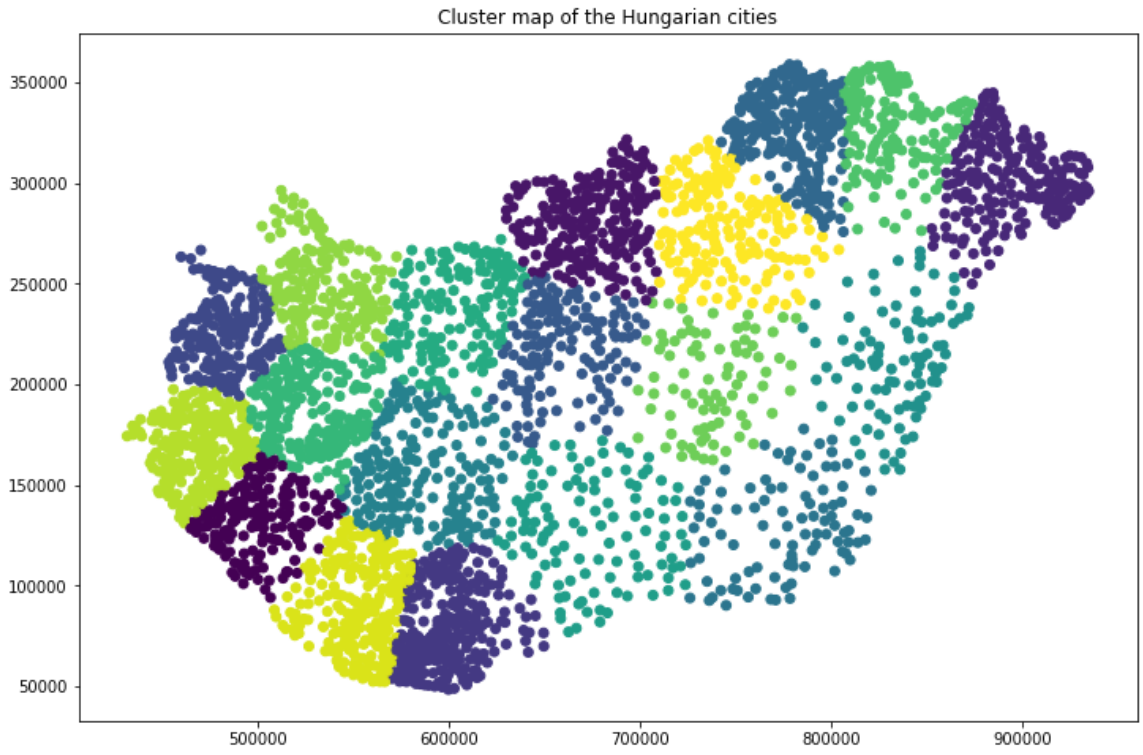
```
pred = KMeans(n_clusters=19).fit_predict(points)
print(pred)
print(len(pred))
```

```
[ 8  3 18 ... 13  2  6]
3147
```

Plot figure:

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize=(12, 8))

# Fetch list of X and Y coordinates
xs = [point[0] for point in points]
ys = [point[1] for point in points]

# Put the cluster points on the plot
plt.scatter(xs, ys, c=pred)

# Display plot
plt.title("Cluster map of the Hungarian cities")
plt.show()
```



Clustering raster images

Read the dataset

The `data/LC08_L1TP_188027_20200420_20200508_01_T1_Szekesfehervar.tif` file is a segment of a Landsat 8 satellite image of Székesfehérvár city, Lake Velence and their surroundings, acquired on 2020 April 20. It should be familiar from Chapter 12 (12_spatial_raster.pdf).

In [5]:

```
import rasterio
szfv_2020 = rasterio.open('../data/LC08_L1TP_188027_20200420_20200508_01_T1_Szek
esfehervar.tif')
print(szfv_2020.count) # band count
print(szfv_2020.width) # dimensions
print(szfv_2020.height)
```

```
11
1057
645
```

Read the red, green blue and NIR bands:

In [6]:

```
blue = szfv_2020.read(2)
green = szfv_2020.read(3)
red = szfv_2020.read(4)
nir = szfv_2020.read(5)
```

Single-band clustering

Cluster the satellite image based on the near-infrared band.

In [7]:

```
nir_1d = nir.reshape(nir.shape[0] * nir.shape[1], 1)
print(nir_1d.shape)
```

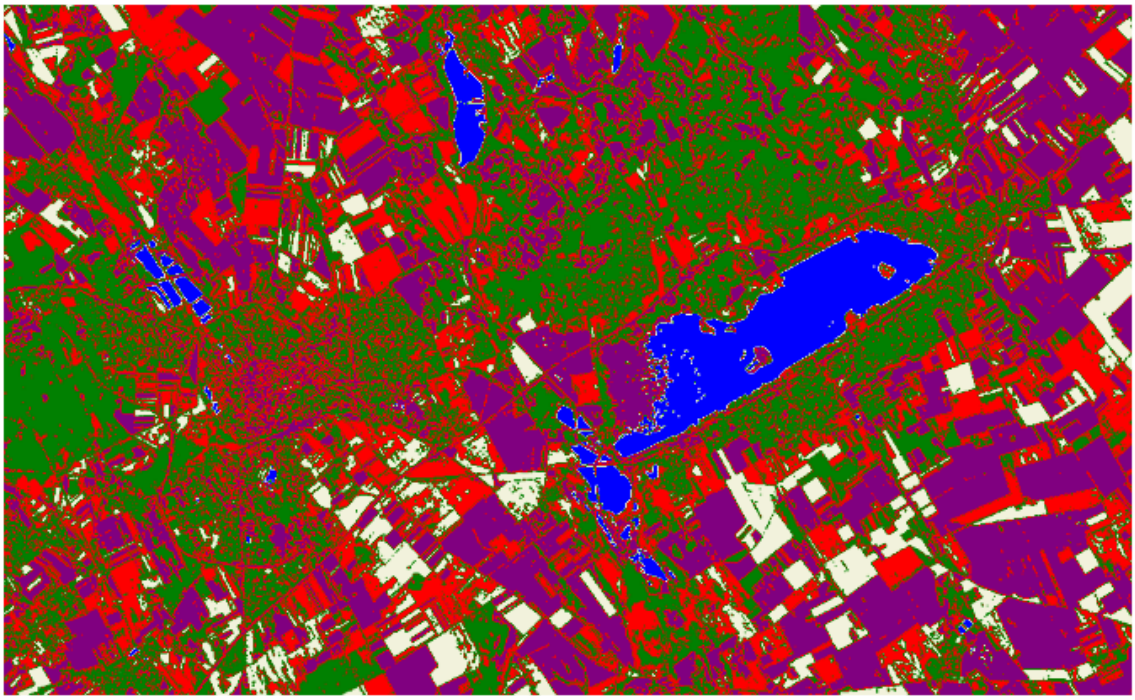
```
(681765, 1)
```

In [20]:

```
pred = KMeans(n_clusters=5).fit_predict(nir_1d)
img_clusters = pred.reshape(nir.shape)
```

In [24]:

```
import matplotlib.colors as mc
cmap = mc.LinearSegmentedColormap.from_list('', ['purple', 'red', 'green', 'beig
e', 'blue'])

plt.figure(figsize=[12,12])
plt.imshow(img_clusters, cmap=cmap)
plt.axis('off')
plt.show()
```



Multi-band clustering

Cluster the satellite image based on the RGBN (red, blue, green NIR) bands.

In [10]:

```
red_1d   = red.reshape(red.shape[0] * red.shape[1], 1)
green_1d = green.reshape(green.shape[0] * green.shape[1], 1)
blue_1d  = blue.reshape(blue.shape[0] * blue.shape[1], 1)

rgbn_1d = [(0, 0, 0, 0)] * (red.shape[0] * red.shape[1])
for i in range(red.shape[0] * red.shape[1]):
    rgbn_1d[i] = (red_1d[i, 0], green_1d[i, 0], blue_1d[i, 0], nir_1d[i, 0])

print(rgbn_1d[10000]) # print random item
```

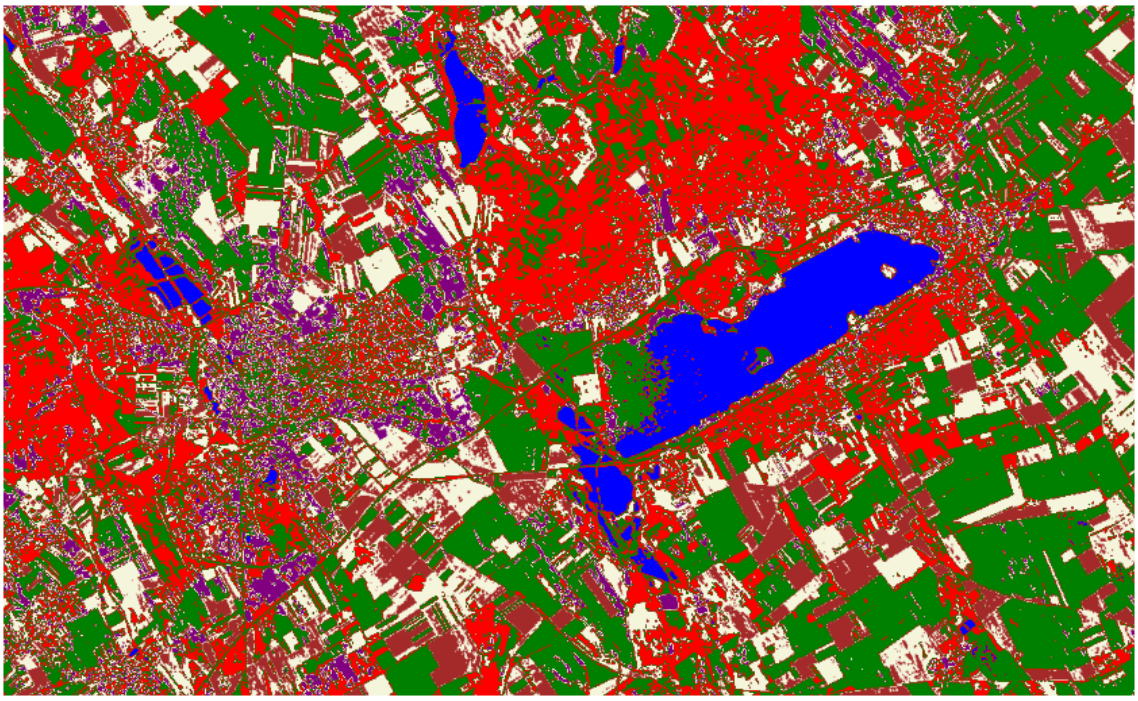
```
(8434, 8678, 9156, 15104)
```

In [28]:

```
pred = KMeans(n_clusters=6).fit_predict(rgbn_1d)
img_clusters = pred.reshape(red.shape)
```

In [32]:

```
cmap = mc.LinearSegmentedColormap.from_list('', ['blue', 'red', 'green', 'brown'
, 'beige', 'purple'])

plt.figure(figsize=[15,15])
plt.imshow(img_clusters, cmap=cmap)
plt.axis('off')
plt.show()
```



Downsampling

The `LC08_L1TP_188027_20200420_20200508_01_T1` file is a complete Landsat 8 satellite image tile, containing Budapest and parts of Western-Hungary, acquired on 2020 April 20.
Download: https://gis.inf.elte.hu/files/public/landsat-budapest-2020 (https://gis.inf.elte.hu/files/public/landsat-budapest-2020) (1.4 GB)

In [13]:

```
import rasterio
bp_2020 = rasterio.open('LC08_L1TP_188027_20200420_20200508_01_T1.tif')
print(bp_2020.count) # band count
print(bp_2020.width) # dimensions
print(bp_2020.height)
```

```
11
7981
8071
```

To speed up processing larger raster files, we may downsample them for the price of reducing the accuracy of the result.

First, define the resampling function:

In [14]:

```
from rasterio.enums import Resampling

def read_resampled_band(dataset, band, resample_factor):
    data = dataset.read(band,
        out_shape=(
            1,
            int(dataset.height * resample_factor),
            int(dataset.width * resample_factor)
        ),
        resampling=Resampling.bilinear
    )
    return data
```

Read the blue, green, read and near-infrared bands into *Numpy* arrays. Resample them to a smaller size to make further processing (clustering especially) faster.

In [15]:

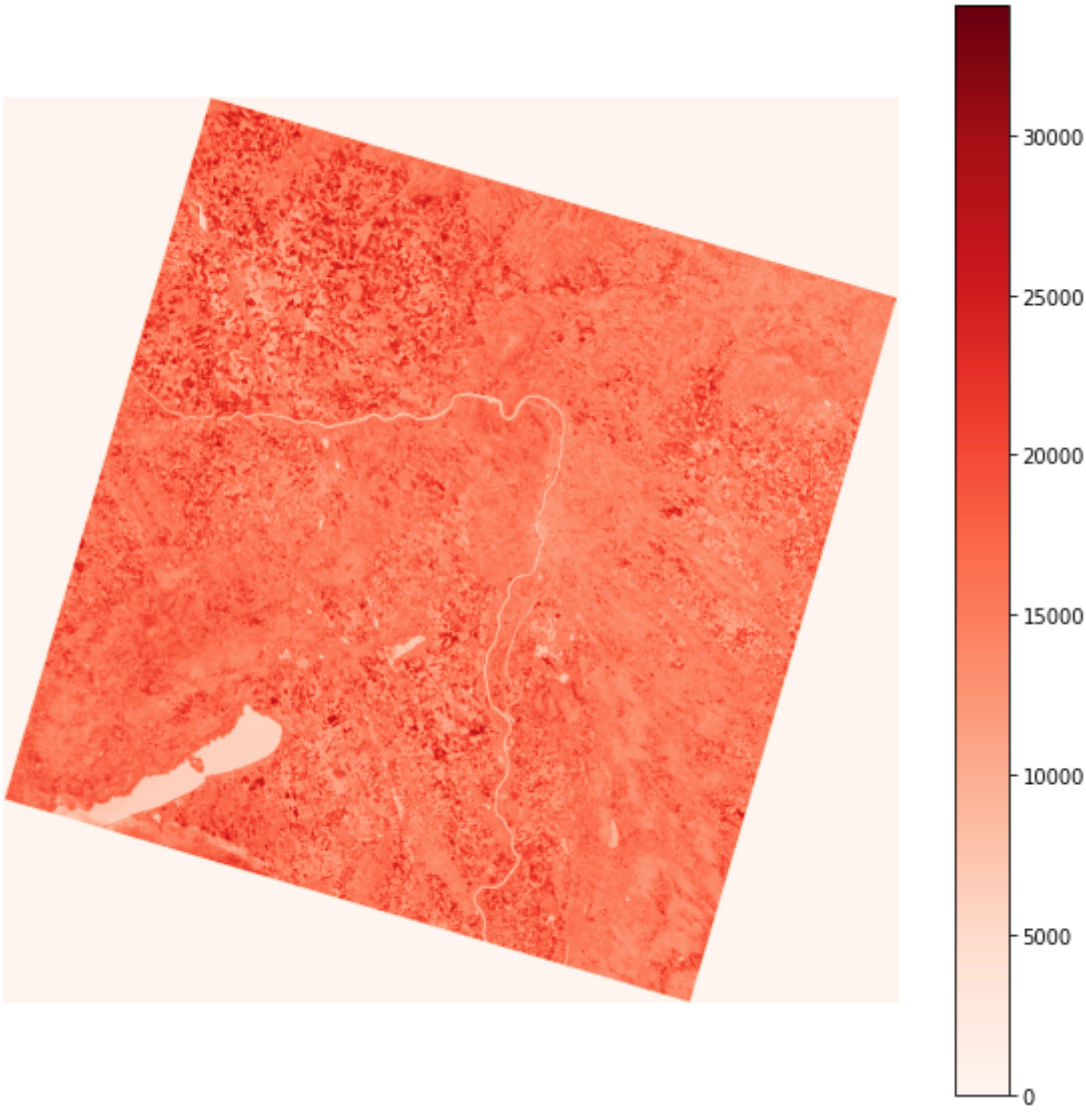
```
bp = {}
bp['blue'] = read_resampled_band(bp_2020, 2, 1/4)
bp['green'] = read_resampled_band(bp_2020, 3, 1/4)
bp['red'] = read_resampled_band(bp_2020, 4, 1/4)
bp['nir'] = read_resampled_band(bp_2020, 5, 1/4)

print(bp['red'].shape)
```

```
(2017, 1995)
```

Display the near-infrared band for verification:

In [16]:

```
plt.figure(figsize=[10,10])
plt.imshow(bp['nir'], cmap='Reds')
plt.axis('off')
plt.colorbar()
plt.show()
```

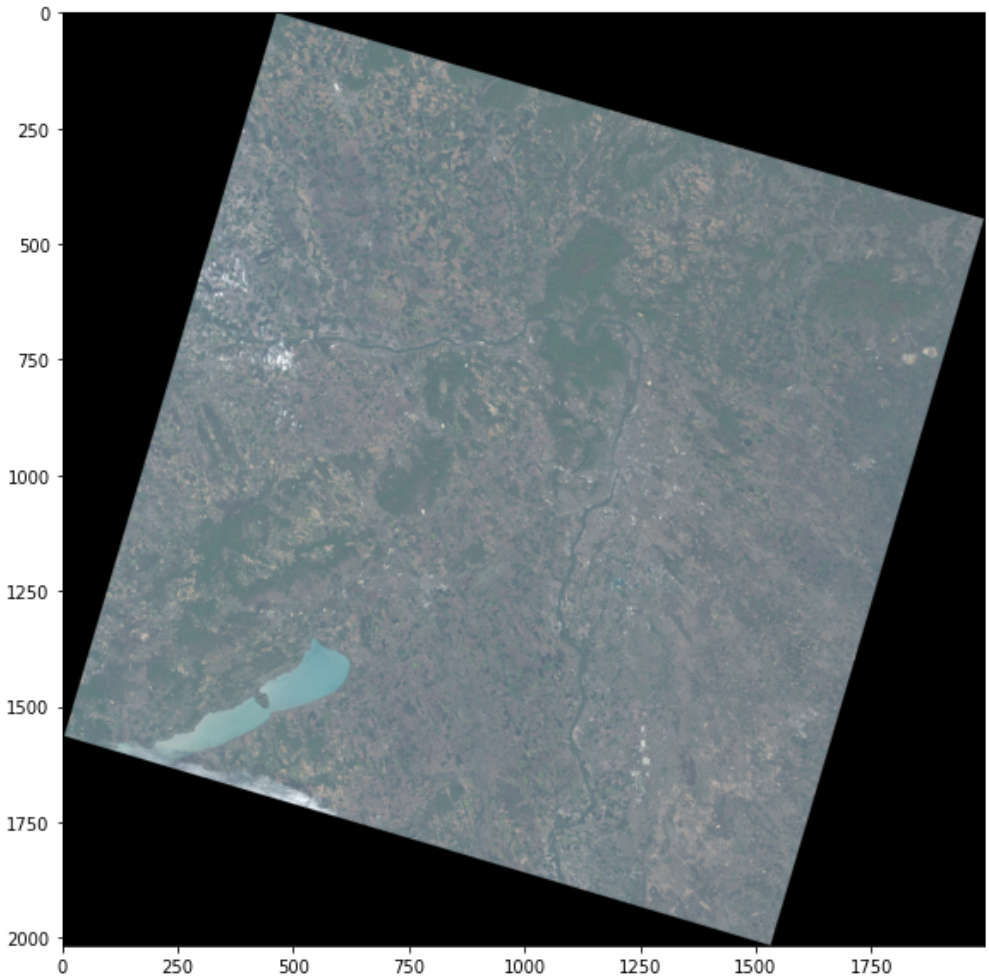


Display the RGB image for verification:

In [17]:

```
from rasterio.plot import show
import numpy as np

bp['red_max'] = np.percentile(bp['red'], 99.99)
bp['blue_max'] = np.percentile(bp['blue'], 99.99)
bp['green_max'] = np.percentile(bp['green'], 99.99)

# astype('f4') is a numpy function to convert to float (4 byte)
bp['redf'] = bp['red'].astype('f4') / bp['red_max']
bp['bluef'] = bp['blue'].astype('f4') / bp['blue_max']
bp['greenf'] = bp['green'].astype('f4') / bp['green_max']
bp['rgb'] = [bp['redf'], bp['greenf'], bp['bluef']]

plt.figure(figsize=[10,10])
show(bp['rgb'])
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data
([0..1] for floats or [0..255] for integers).
```



Summary exercise on clustering

Implement a functions which performs single band clustering on a *rasterio* band (*NumPy* array). Execute it on the NIR band of the complete satellite image.

Example on how it shall work:

```
single_band_clustering(bp['nir'], ['red', 'black', 'gray', 'green', 'whit
e', 'blue']) # 6 clusters with these colors
```

In [18]:

```
def single_band_clustering(band, clusters=['red', 'black', 'gray', 'green', 'whi
te', 'blue']):
    band_1d = band.reshape(band.shape[0] * band.shape[1], 1)

    pred = KMeans(n_clusters=len(clusters)).fit_predict(band_1d)
    img_clusters = pred.reshape(band.shape)

    cmap = mc.LinearSegmentedColormap.from_list('', clusters)

    plt.figure(figsize=[12,12])
    plt.imshow(img_clusters, cmap=cmap)
    plt.axis('off')
    plt.show()
```

In [19]:

```
single_band_clustering(bp['nir'])
```