University of Southern Queensland

Project Part 3 – Final Project Report

CSC8004 – Data Mining

Rudiger von Hackewitz

Due Date: Friday, 15 June 2018

Table of Contents

List of Figures

List of Tables

List of Code Snippets

List of Equations

1. Introduction

KMeans clustering has been in use in the IT industry for a long time. In the last decade, with the advent of strong computer power and larger data sets, the power and usefulness of this algorithm became more evident.

When I started research about use cases for KMeans clustering, I realised that this algorithm is used extensively in image clustering. This raised my interest, and after reading several articles about image processing, I stumbled across the paper *Image Segmentation via Improving Clustering Algorithms with Density and Distance* [1] by Zhensong Chen et al.

The paper discusses a new approach for image clustering, using Density Peaks (short: DP) for segmentation of images. The segmentation results of this new approach are then compared with classic image segmentation processes such as KMeans.

In my project, I implemented the newly proposed DP algorithm and compared clustering results against the standard KMeans clustering approach. All work has been developed in Python 3[2], without the use of advanced mathematical or machine-learning libraries, of which there are plenty available in Python 3.

It is highly recommended to read the paper [1] by Zhensong Chen et al. first to get a full understanding about the DP algorithm; only then the reader should proceed with the reading of this project report. The paper by Zhensong Chen et al. is repeatedly referred to in my document.

In this project report, I offer details about my implementation of the DP and KMeans implementations, and run those two processes against personal images from a holiday in New Zealand in January 2018. Plenty of image segmentation results with comparisons and various hyper-parameter settings are included in this final project report.

Some focus is given to the runtime performance issues that I encountered during coding and testing. I developed a grid of mini-cubes with summarised pixel data to overcome those runtime problems.

The report includes details about a GUI application, how it can be installed, configured and run to compare both clustering algorithms (DP and KMeans) side-by-side in a single application window.

Further, I document some of the interesting differences and similarities between DP and KMeans clustering and will comment on some of the findings by Zhensong Chen et al. in their paper *Image Segmentation via Improving Clustering Algorithms with Density and Distance* [1].

In a reflection I will highlight some of the lessons that I learnt while working on this project.

Finally, an outlook will include some of the new questions and research topics that may be covered in a future research paper.

2. System Architecture

2.1. Overview

Due to performance issues, the KMeans and Density Peak (DP) clustering algorithms required pre-processed image data. The image segmentation was not undertaken on the original image, but on an image with largely reduced pixel numbers (or data points).

The Density-Peak (short: DP) and KMeans clustering are then completed on the pre-processed images with largely reduced distinct data points (pixels).

A GUI interface has been developed which allows the user to display the clustered images in an application window. Optionally, a copy of clustered images can also be stored in a log folder of the operating system.

2.2. Architecture Diagram

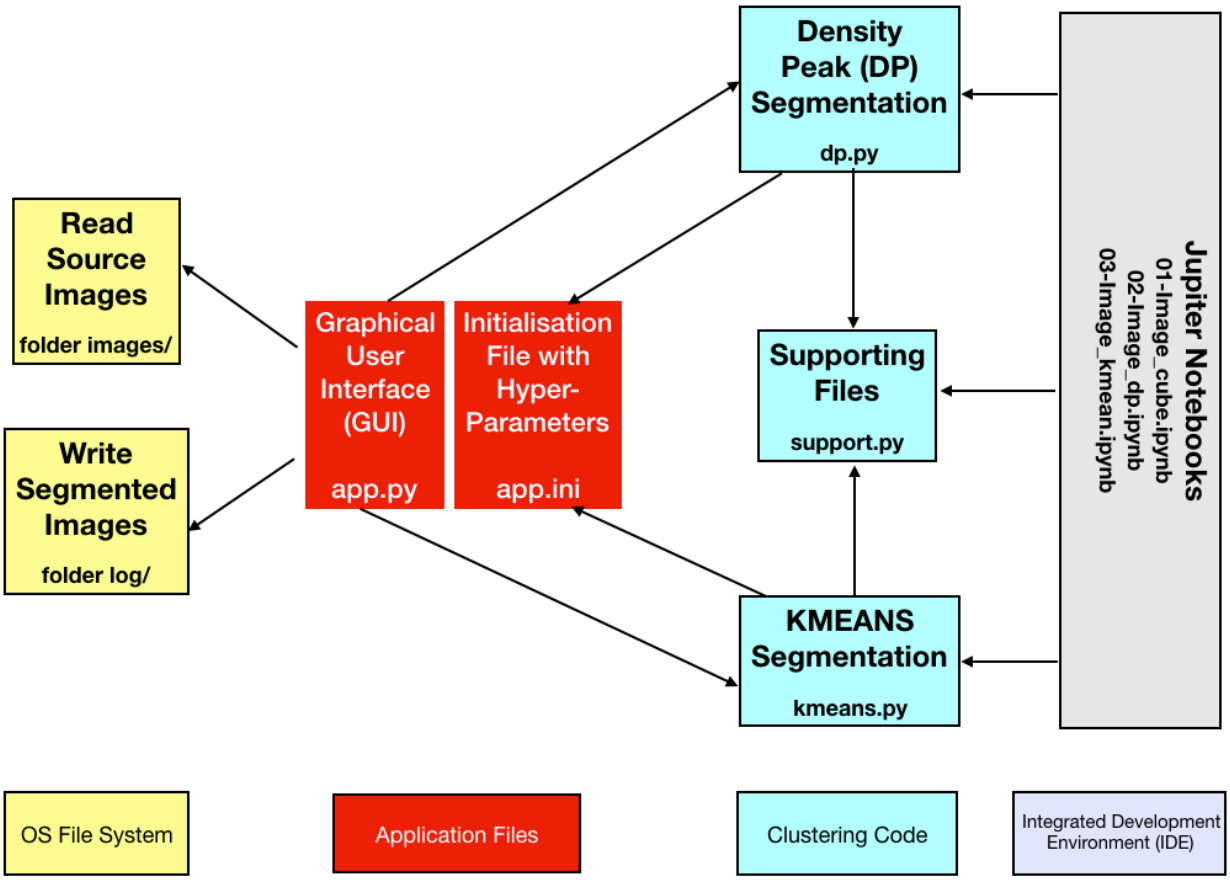


Figure 2.1 Architecture Diagram

The two files in the red-shaded rectangles are the 'heart' of the GUI application. The grey area to the right is the Integrated Development Environment (IDE) and not required to run the application.

Please note: the application needs read access to the sub-folder images/ and write access to sub-folder log/. If required, permissions should be set accordingly on the operating system level.

The code for segmentation of images (DP and KMeans) is depicted in the light blue boxes. The code is encapsulated in separate Python classes and is called by the graphical user interface.

2.3. Python Code

2.3.1. Libraries

The following ten Python libraries have been used for the development of the application; reasons for their use are provided in the table below.

Python Library	Reason
configparser	Used to read in parameter settings from initialisation file app.ini
datetime	Helps measure elapsed runtime for image segmentation processes for DP / KMeans, and with various hyper-parameter settings for runtime comparisons
functools	Function 'reduce' in this library is used to simplify (collapse) iterative operations across lists in Python
math	Functions log, pow and exp from this library are required for various calculations.
matplotlib	Plotting graphs and images in Jupyter Notebook (not required for running the app)
os	Used for interaction with OS file system (reading / writing files)
PIL[5]	PIL (short for: Python Imaging Library) used for reading and writing images and their pixel streams
random	Used for initialising randomly the cluster centres in the KMeans algorithm before starting the segmentation process
tkinter[4]	Python library for GUI development and used in Python file app.py
xlrd	Library required to load test data from Microsoft Excel into Python (only required for system testing in Jupyter Notebook, but not for running the application app.py)

Table 2.1 Python Libraries Used in Application and Jupyter Notebook

The ten Python libraries are all included in a Python 3 standard installation; I have installed Python 3 on my McBook Air through the Anaconda distribution [13].

None of the popular Python libraries such as numpy[15] or scikit-learn[16] have been used to develop the image segmentation processes in this project. All code has been built from scratch with standard Python functionality.

2.3.2. Modules

To follow is the list of source files, modules and Python classes that have been used and developed for this application.

File Name or Directory	Description
images/	This directory includes seven source file images that have been tested by the program.
log/	This directory includes the clustered images. This is optional and logging of clustered images can be set in the application GUI via a checkbox.
app.ini	Application initialisation file – for details see 4.5 – Initialisation File app.ini
app.py	Contains Python class *Application* to start the GUI application; includes all graphical controls to allow the user to manage image segmentation processes

File Name or Directory	Description
dp.py	Class *DPPoints* implements image pre-processing and the DP clustering algorithm; class *DPImage* is derived from this class and implements image-specific methods
kmeans.py	Class *KMeansPoints* implements image pre-processing and KMeans clustering process for pre-processed images; class *KMeansImage* is derived from *KMeansPoints* and implements image-specific methods
support.py	Source file with supporting functions for distance measures and retrieval of parameter values from the app.ini initialisation file

Table 2.2 Application Modules and Folders

Please note: directory 'testdata/' and the Jupyter Notebook files *.ipynb are complementary files and provided so that the reader can re-run the code with some of the test data and re-produce output with graphs as they are included in this report.

3. Image Segmentation Algorithms

3.1. Overview

When I developed the proof-of-concept for Part 2 (Progress Report for this project), I followed the standards that are used for implementation of most clustering processes: read in all data points (aka pixels) as data stream and process the information based on the RGB value for each pixel (in sequential order).

This approach allowed me to achieve successful KMeans clustering for images, though it was very slow (run times up to several minutes, in some cases even hours)!

Once I had completed the implementation of the Density Peak (DP) clustering algorithm, it turned out that I had reached a dead end with my code: not a single DP segmentation process could complete within hours of running!
I started researching alternative Python libraries, but they all had the same problem: once faced with the processing of 100,000+ data points (aka pixels), we can't just stream the data any longer to perform basic calculations, such as: find the maximum possible distance between any two points in the set, or find the closest pixel for each pixel in the image. All these operations are of the order $O(n^2)$ and cannot be completed by any program in reasonable timeframes.

After hitting this dead end, I thought about alternative options and it became clear that two things would help to solve this issue:

1. Somehow the program should use indexed data points to improve access to relevant data (pixels) without the need to loop through all data records (in relational database terms this is a 'full table scan' vs 'indexed data access')
2. Meaningful reduction of data points for processing, without compromising on the quality of the image too much

I conceived the following two approaches for implementation of those two ideas:

- Ad 1: It was beyond the time and scope of this project to set up a relational database schema for the image pixels. However there is the concept of Dictionaries[14] available in Python. It uses a hash-key to access data for a particular key (or index) and offers excellent performance. I have decided to convert all list loops, eg to find an attribute for a particular pixel, into dictionaries. This offers significant improvements for the overall KMeans and DP clustering algorithms.
- Ad 2: Rather than working with individual pixels, I have divided the RGB cube (with length 256) into several sub-cubes of equal length (e.g. cubes with length of the geometric numbers of 2. For each mini-cube within the overall RGB cube, I have then counted the number of pixels that sit within the mini-cube, and assigned this number to the pixel in the centre of the mini-cube. Completing this process for an image with 100,000+ pixels, allowed me to compress the number of distinct data points (pixels) below 1,000 in most cases!

With the 2 steps described above, the structure of the input data for the clustering processes had been significantly changed, and the traditional KMeans algorithm did no longer work. Code became far more complex (for example, the KMeans implementation increased from about 50 lines to over 150 lines of code!) However, the results were astounding: KMeans processes, that ran for 10 min and longer could now be completed in under 5 sec!

The approach was a compromise: giving away a little bit of image quality allowed me to achieve very good runtimes for KMeans and DP clustering algorithms.

On a high level, I ended up with the following process flow for the segmentation of images:

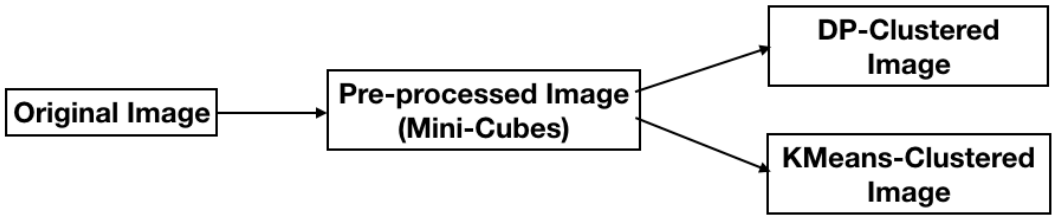


Figure 3.1Overall Image Segmentation Process

In the following sections I will explain in more detail each of the three image processing steps.

3.2. Image Pre-Processing

In the first step, I started to slice the RGB Pixel space (256x256x256) into several mini-cubes with equal length. To achieve equally sized mini cubes, I had to choose geometric numbers of 2 (e.g. 1, 2, 4, 8, up to 256).

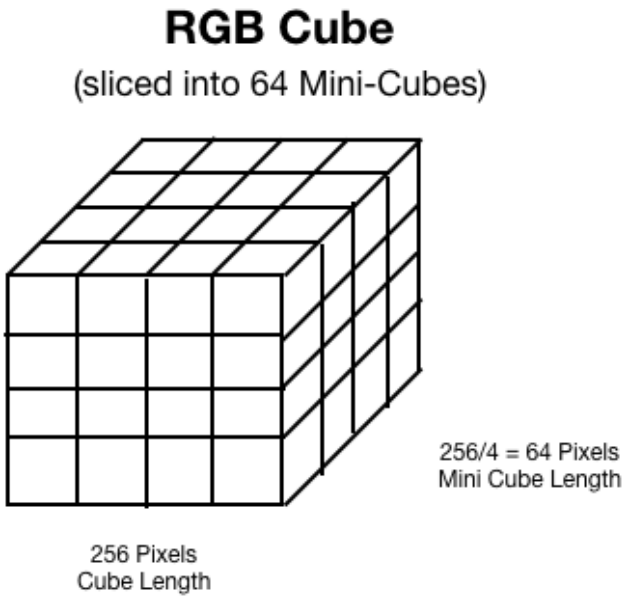


Figure 3.2 Sliced RGB Cube Image Space

In the next step I have counted the number of image pixels in each of the mini-cubes and assigned their number to an artificial point at the centre of the cube:

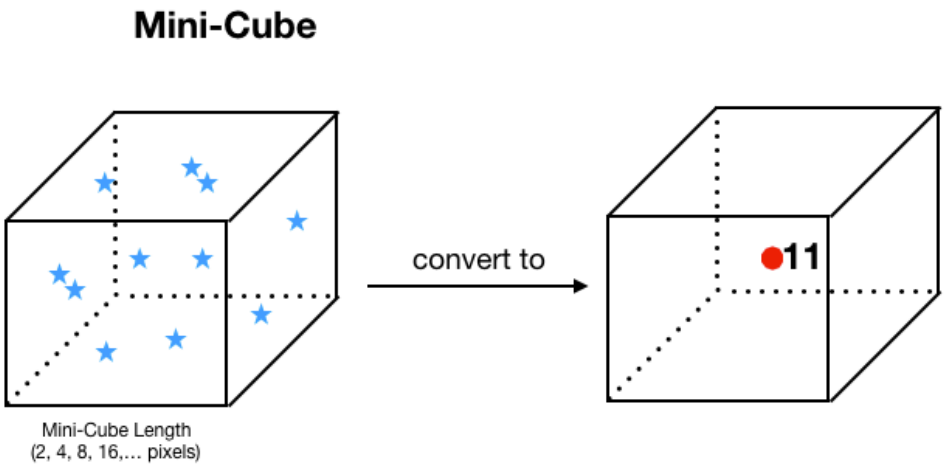


Figure 3.3 Allocation of Pixels in Mini-Cube to its Centroid

This pre-processing step is very fast (of the order O(n). The code extract for this step is provided below:

```
# preprocessed image data (to improve runtimes of clustering algorithm)
def get_pre_processed_data (self):
    return [self.p_map[p] for p in self.points]


def pre_process_points (self, data):
    self.points = data
    self.pnts = dict()
    self.p_map = dict()
    for _pp in self.points:
        pp = list(_pp)
        p = tuple(map(lambda x: ((int(x/self.GRANULARITY))*self.GRANULARITY) + int(self.GRANULARITY/2), list(pp)))
        if p not in self.pnts:
            self.pnts[p] = 1
        else:
            self.pnts[p] = self.pnts[p] + 1
        if _pp not in self.p_map:
            self.p_map[_pp] = p
```

Code 3.1 Pre-Processing of Image into Mini-Cubes

These 15 lines of code were the 'life saver' for my project, as it allowed me to drastically reduce the number of pixels that had to be processed for the KMeans and DP clustering algorithms.

I have executed the code, with length from 1 through to 256 (all geometric numbers of 2). The result, with images of nine different granularity levels, is displayed in the diagram below.

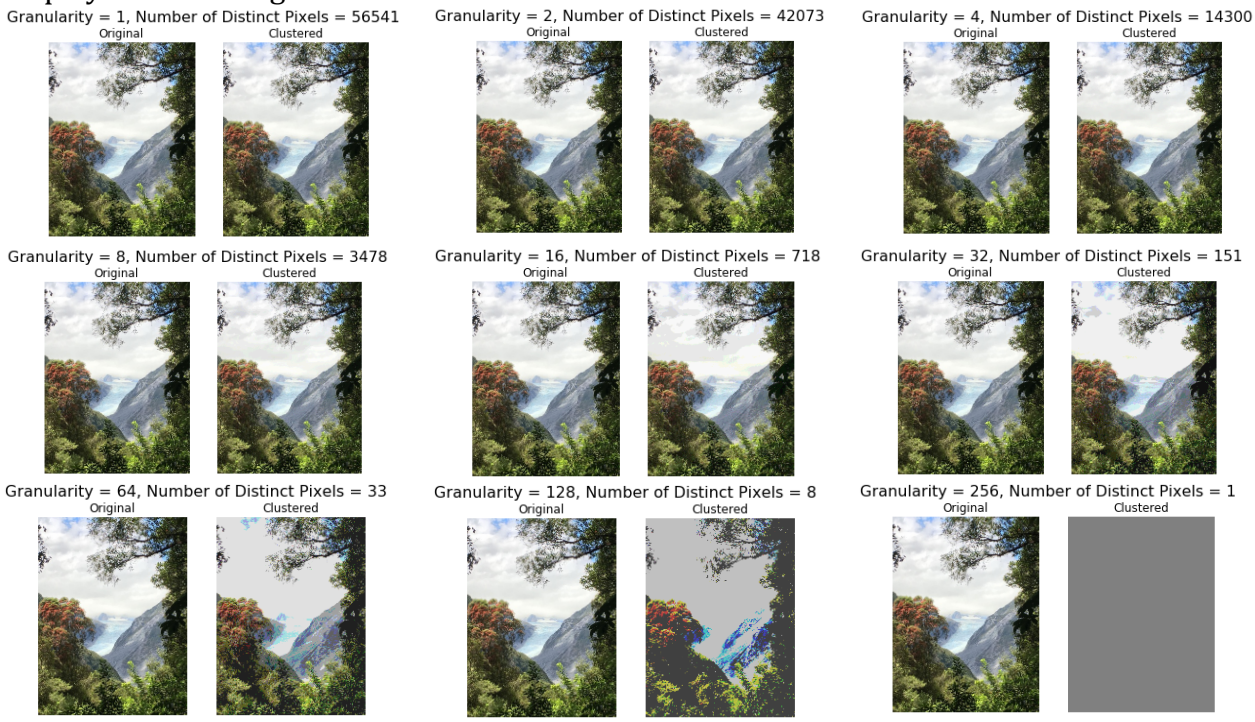


Figure 3.4 Quality of Pre-Processed Images for Various Mini-Cube Sizes

The output in this graph can be reproduced in *Jupyter Notebook 01-Image_cube.ipynb*.

With granularity = 1, there is no difference between the original and the pre-processed image. With granularity = 256, only one large cube remains in the RGB space. Its centre is (127,127,127) and this is exactly the colour that is shown for the pre-processed image (grey).

It is remarkable that cube size 128 produces a total of 8 clusters (2^3), which in itself could be considered as a reasonable segmentation of the initial image with a fixed number of 8 segments!

Final decision: we achieve reasonable image quality in the pre-processed image with a mini-cube size (granularity) of 16. I used this as standard value for all image-clustering processes. Please note: the value is configurable in the app.ini file, in the global section and can also be altered in the GUI interface.

3.3. Distance Metric

The distance metric is relevant for the KMeans and DP clustering algorithms. I have implemented the distance metric as a configurable parameter, and the code supports the Supremum, Euclidean and Manhattan distance measures. Implementations are based on the more general Minkowski distance:

$$d_{Minkowski}(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{1/p} \quad with\ p \geq 1$$

Equation 1 Definition of Minkowski Distance

With p=1, the Minkowski distance becomes the Manhattan distance, and with p=2, it turns into the Euclidean metric.
From a mathematical point of view, it is interesting to note that the Minkowski distance becomes the Supremum distance with $p \rightarrow \infty$

The distance metric is implemented in file support.py and the code extract is provided below.

```
# set up the function for the minkowski distance, which is later used for calculation
# of Euclidean and Manhattan distance
def dist_minkowski(point1, point2, p):
    return pow(sum([pow(abs(point1 - point2),p) for point1, point2 in zip(point1,point2)]), 1/p)

# calculate the distance between two points
# choose different function for Euclidean, Manhattan and Supremum:
def dist_func(point1, point2, dist):
    if dist == 'Euclidean':
        return dist_minkowski(point1, point2, 2)
    elif dist == 'Manhattan':
        return dist_minkowski(point1, point2, 1)
    elif dist == 'Supremum':
        # return dist_minkowski(point1, point2, 'infinite') can't be implemented as calculation, but is a limes
        # therefore, a native calculation is used for the 'Supremum' distance
        return max([abs(point1 - point2) for point1, point2 in zip(point1,point2)])
    else:
        raise Exception('Distance function '+dist+' is not defined in program')
```

Code 3.2 Code for Calculation of Distance Metric, based on Minkowski Distance

For the images tested in this project, all three metrics produce similar results. The three metrics are visualised in the following two-dimensional graph, which reduces the three dimensions of the colour components to a flat plane:

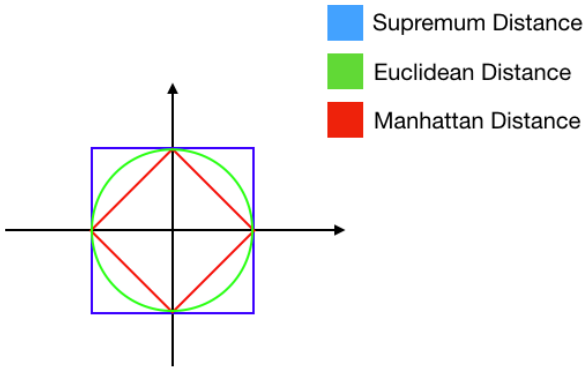


Figure 3.5 Illustrating Areas of Equal Distance from Origin for Distance Metrics

The blue line describes points of equal distance to origin for the Supremum distance; the green and red line does the same for the Euclidean and Manhattan distance, respectively.

The blue rectangle should be thought of as cube in the RGB space and the green circle should be thought of as sphere.

Based on the illustrations above, it might be the most logical choice to work with the Supremum distance, as the RGB space has already been divided into many mini-cubes. However, it turns out that the three distance metrics produce similar and equally good clustering results.

Going forward – if not noted otherwise – I work with the Euclidean metric for image clustering. This metric is set as the default metric in the application interface, but can be changed by the user to Manhattan or Supremum.

3.1. Density Peak (DP) Image Segmentation

3.1.1. Estimation of Average Distance between pixels in a Mini-Cube

All pixels in a mini-cube are now compressed into the centre of its cube. This means that the algorithm will calculate a distance of 0 for all data points in the same mini-cube. However, this is not correct and we use the estimated average distance between two points in a cube.
Rather than mathematically calculating this mean, I have decided to run a simple simulation to estimate the mean differences for Euclidean, Manhattan and Supremum distances. Those values are then used in the density calculation for all pixels in the DP algorithm.

```
from random import random
from math   import sqrt
e=0.0
m=0.0
s=0.0
n=1000000
for i in range(n):
    e = e + sqrt(pow(random() - random(),2) + pow(random() - random(),2) + pow(random() - random(),2))
    m = m + abs(random() - random()) + abs(random() - random()) + abs(random() - random())
    s = s + max(abs(random() - random()),abs(random() - random()),abs(random() - random()))
print("Average Manhattan Distance: "+str(m/n))
print("Average Euclidean Distance: "+str(e/n))
print("Average Supremum Distance:  "+str(s/n))
```

```
Average Manhattan Distance: 0.9995685405040539
Average Euclidean Distance: 0.6616659699696334
Average Supremum Distance:  0.5428835380359948
```

Code 3.3 Simulating Average Distance of 2 Random Points in a Mini-Cube

For the implementation of the DP algorithm, I use the following average distance measure for the points in a mini-cube:

$$d_{Supremum} = 0.54 \times MiniCubeLength$$
$$d_{Euclidean} = 0.66 \times MiniCubeLength$$
$$d_{Manhattan} = 1.00 \times MiniCubeLength$$

Equation 2 Calculation of Average Distance for Pixels in a Mini-Cube

3.1.2. Estimation of Hyper-Parameter DC

In the paper [1] by Zhensong Chen et al., it is recommended to set the scaling factor DC for the Gaussian reach to 0.5% of the maximum distance between all pixels in an image.

Considering the pre-processing of data into the centres of multiple mini-cubes, this parameter was no longer suitable:

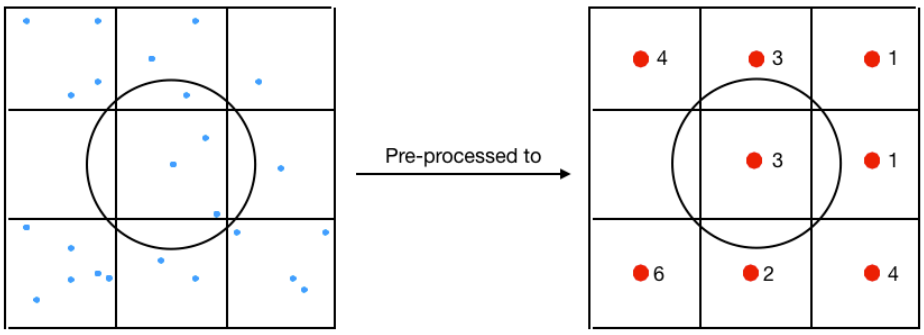


Figure 3.6 Impact of Hyper-Parameter DC on Pixels in Pre-Processed Image

The radius (aka sphere in RGB) for DC (assuming the Euclidean distance), does only reach a portion of the original image pixels, as all pixels are moved further apart by the cube length. As consequence, I had to slightly increase the factor for DC and it turned out, that for a cube length of 16, the value of 3% of the maximum distance between all pixels produces good results, which I achieved with the following formula:

$$DC = \frac{\max_{i,j} d(x_i, x_j)}{DensityScaling} + \log\left(Number\ of\ pixels\ in\ image\right)$$

Equation 3 Calculation of Density Scaling Factor DC for DP Clustering

DensityScaling is a configurable parameter in app.ini and produced good results with the setting *DensityScaling* = 400.0.
With very large number of pixels in an image, the DC factor is slightly 'pushed up'.
The code snippet for the implementation of the DC calculation is provided below.

```
# get the maximum distance between any two pixels in the image
self.max_dist = max([dist_func(p, q, self.dist) for p in self.pnts.keys() for q in self.pnts.keys()])
# set density scaling factor
self.dc = self.max_dist / self.D_SCALING
# take number of records into consideration in the order of natural logarithm:
self.dc = log(float(len(self.points))) + self.dc
```

Code 3.4 Setting of the Density Scaling Factor in the DP Algorithm

3.1.3. Outlier Detection

After mapping the density (rho) and distance (delta) pair into a scatter plot, we find several outliers in the graph that need to be determined by the algorithm. The following graph is taken from Jupyter Notebook *02-Image_dp.ipynb* and shades the calculated outlier region (grey rectangle):

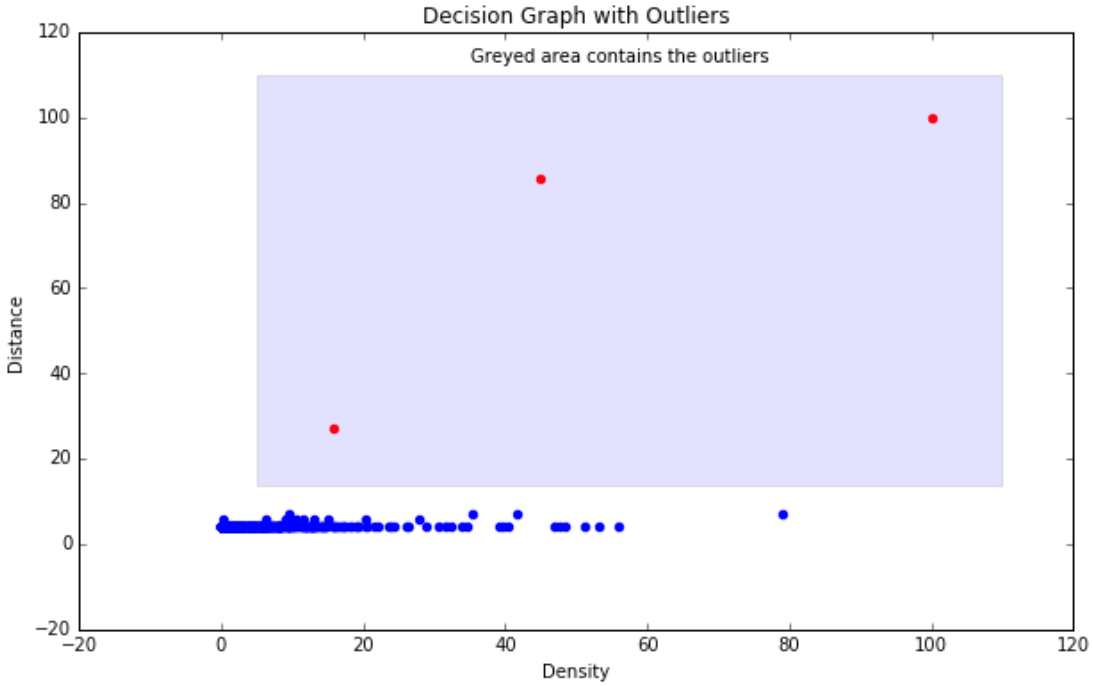


Figure 3.7 Outlier Detection in the Decision Graph of the DP Algorithm

In the diagram above, the outliers are marked in red. I have scaled the Distance / Density values to be in the range from 0 to 100. The following thresholds produced good results for the outlier detection of image pixels:

Density Threshold (Outlier Calculation)
I achieved good results with a density threshold of 5 (or 5% of density range): only pixels above this density threshold are considered as candidates for the cluster centroids.

$$DensityThreshold = DensityRange \times DensityPercentage$$

Equation 4 Calculation of Density Threshold in DP Algorithm

This function is implemented in the following code snippet:

```
# get density threshold for outliers
def get_dens_threshold (self):
    return self.DENSITY_MIN*self.DG_SCALING
```

Code 3.5 Calculation of Density Threshold in DP Algorithm

In the app.ini file I work with the settings *DensityRange* = 100 and *DensityPercentage* = 0.05

Distance Threshold (Outlier Calculation)
I modelled the distribution of the distance (y-axis) via the exponential distribution, tested several settings and achieved good results with the following equation:

$$DistanceThreshold = -ln(PctOutlier) \times Average(dist) + ln(Number\ of\ all\ pixels)$$

Equation 5 Calculation of Distance Threshold in DP Algorithm

PctOutlier is a configurable hyper-parameter in file app.ini and works well with the setting 0.20.
The term $ln\ (Number\ of\ all\ pixels)$ has been added as a correction number for very large images: in those circumstances the chances increase that too many pixels are above the distance threshold, and this is tackled by pushing the distance threshold slightly up when the number of pixels in an image is very large.
The function for the calculation of the distance threshold is implemented in the following code snippet:

```
# get distance threshold for outliers
def get_dst_threshold (self):
    # calculate lambda as the reciprocal value of the emperical mean of the distance values for each record.
    # based on the model that distance follows an exponential distribution
    lambd = len(self.dst.keys()) / sum(self.dst.values())
    self.dst_threshold = -log(self.PCT_OUTLIER)/lambd
    return self.dst_threshold + log(len(self.pnts.keys()))
```

Code 3.6 Calculation of Distance Threshold in DP Algorithm

3.1.4. Assigning Remaining Pixels to Cluster Centres

The recursive function '*assign_point*' has been developed to assign the remaining pixels to their appropriate groups:

```
# now assign the remaining points, by building up the dictionary assigned_group in recursive calls:
def assign_point(self, p):
    if p not in self.assigned_group:
        # if not yet assigned, then find the closest point with higher density:
        q = min([pp for pp in self.pnts.keys()
                    if self.dens[pp] > self.dens[p]], key = lambda x : dist_func(x, p, self.dist))
        # call the same function recursively to find appropriate group for this point:
        self.assigned_group[p] = self.assign_point(q)
    # return the appropriate pixel centroid for the point
    return self.assigned_group[p]


# assign points to relevant groups by calling function assign_group for each key in the dictionary
def assign_remaining_points (self):
    # first initialise the dictionary with the outliers. They are centroids of their respective clusters
    for p in self.centroids:
        self.assigned_group[p] = p
    # then assign each remaining point in the dictionary to their appropriate group
    for p in self.pnts.keys():
        self.assign_point(p)
```

Code 3.7 Assigning the Remaining Pixels in the DP Algorithm

The recursive function *assign_point* has very high recursive depth in the beginning of the process, when only very few pixels have been assigned to groups. However, as the process gradually works through all the image pixels, less and less recursions will be required. It is remarkable that this (relatively) complex function can assign all remaining points in order close to *O(n)* and causes no performance issues.

3.2. KMeans Image Segmentation

The KMeans clustering algorithm works with some slight modifications, caused by the different structure of the input data.

Each input record (aka centre of a mini-cube) has been assigned a weight, or number of pixels it represents. Based on this weight the calculation for the distance between such a point and the centroid is multiplied by this 'weight'.

The KMeans algorithm is implemented in file kmeans.py and lacks some of the charm and conciseness of the general code for KMeans clustering. The original implementation from less than 50 lines has now increased to 100+ lines of code. To follow are the key lines of the code that calculates the centroids of the KMeans clusters (part of class KMeansPoints):

```
# adds the values of two lists for each dimension
# for example:  sum_dimensions([1,2,3],[4,5,6]) = [5,7,9]
def sum_dimensions(self,lst1,lst2):
    return [lst1_i + lst2_i for lst1_i, lst2_i in zip(lst1, lst2)]


# calculate the mean value of the lists across all dimensions
# (which are the 3 RGB channels for images)
# input is a list of points (lists)
# for example: mean_func([[1,2,3],[4,5,6]]) = [2.5, 3.5, 4.5]
def mean_func(self,lists, i):
    l = sum([self.pnts[tuple(p)] for p in self.pnts if self.assigned_group[p] == i])
    if l > 0:   # avoid division by zero
        # sum up across all dimensions
        vs = reduce(self.sum_dimensions, lists)
        # divide by the number of points to get the mean across each dimension
        return [(1/l) * v_i for v_i in vs]
    else:
        return 0; # consider the mean to be zero when the list is empty

# last not least, run all steps of the KMeans algorithm in the single function call 'run' and measure the time
def run(self, data):
    self.points = [tuple(p) for p in data]
    # initialise data
    self.initialise()
    t1 = datetime.now()
    # preprocess the pixels
    self.pre_process_points(data)
    # pick k random points to start with the process and assign points to groups accordingly
    if self.SET_RANDOM_SEED:
        random.seed(self.k * len(self.points))
    rnd  = random.sample(self.pnts.keys(), self.k)
    self.new_means = [list(r) for r in rnd]
    self.means = list()
    self.prnt_sum = dict()

    # assign groups accordingly to the centre they are closest to
    for p in self.pnts.keys():
        self.assigned_group[p] = min([i for i, _ in enumerate(self.new_means)],
                                     key = lambda x: dist_func(list(p), list(self.new_means[x]), self.dist))
        self.prnt_sum [p] = list([i * self.pnts[p] for i in list(p)])

    # break, once we don't get any changes any more in the cluster means
    while self.means != self.new_means:
        self.counter = self.counter + 1         # increment counter by 1
        self.means = self.new_means
        # Recalculate the k centroids, based on the new data
        self.new_means = [self.mean_func([self.prnt_sum [p] for p in self.pnts.keys() if self.assigned_group[p] == i], i)
                          for i in range(self.k)]

        # assign groups accordingly to the centre they are closest to
        for p in self.pnts.keys():
            self.assigned_group[p] = min([i for i, _ in enumerate(self.new_means)],
                                         key = lambda x: dist_func(list(p), list(self.new_means[x]), self.dist))
    t2 = datetime.now()
    self.seconds = (t2-t1).seconds
```

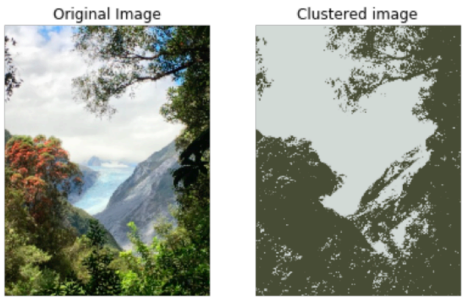
Code 3.8 Implementation of KMeans Algorithm

With the generic KMeans algorithm, based on partitioned mini-cubes, we can now derive class KMeansImage for the clustering of pixels in an image:

```
#
# now derive a separate class for the kmeans algorithm on images. It is basically 'wrapped around'
# the base class KMeansPoints and adds some specific features for the processing of pixels (as
# some specialised kinds of 'points'. In particular: pixels need to be returned in the correct order
# and the means of pixels need to be integers (rather than floats).

from PIL import Image
# https://pillow.readthedocs.io/en/5.1.x/reference/Image.html

class KMeansImage(KMeansPoints):
    def __init__(self, k, dist='Euclidean',preprocessing=True):
        KMeansPoints.__init__(self, k, dist, preprocessing)
        self.image = list()

    def pre_process_img (self, image):
        self.image = image
        self.points = list(image.getdata())
        KMeansPoints.pre_process_points(self, self.points)

    def run_img(self, image):
        self.image = image
        self.points = list(image.getdata())
        KMeansPoints.run(self, self.points)
    def get_data_img (self):
        # now convert to an integer
        rmeans = [list(map(int,m)) for m in self.means]
        return [tuple(rmeans[self.assigned_group[self.p_map[p]]]) for p in self.points]
```

Code 3.9 Class KMeansImage Derived from Class KMeansPoints

Performance differences are striking: the standard KMeans algorithm, which was implemented for the progress report in Part 2 of this project took 7 seconds for the following image:

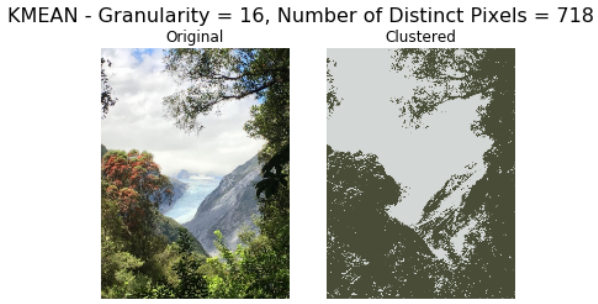



```
print("Clustering the image with k="+str(k)+' and '+ dist +' distance \n '
                          +' took '+str(km.seconds)+' seconds, with ' +str(km.counter)+' loops')
print('pixels:  '+str(len(km.points)))
```

```
Clustering the image with k=2 and Manhattan distance
  took 7 seconds, with 7 loops
pixels:  139968
```

Figure 3.8 KMeans Clustering on Raw Image Pixels without Pre-Processing

With the revised code in class KMeansImage, KMeans clustering of same image with the same parameter settings can be completed in less than 1 second!




```
Granularity:                    16
Number of pixel clusters:       718
Number of centroids:            2
Total Number of pixels in image: 139968
Number of loops:                4
Runtime (in seconds):           0
```

Figure 3.9 KMeans Clustering on Pre-Processed Images with Cube Size 16

Code and output for the above graph are provided in Jupyter Notebook *03-Image_kmean.ipynb*.

Overall - performance of the redeveloped KMeans algorithm, which is based on pre-processed 'image cubes', runs magnitudes faster than the standard KMeans clustering process. Some processes took previously several minutes (in particular with the Euclidean metric and higher values for k). With the new KMeans implementation, those processes can be completed in a few seconds!

The reader is encouraged to test the KMeans and DP algorithms in the GUI implementation for this project (app.py). Several test images, including the ones used in this notebook, are included in the submission. The reader may also test the process with his or her own images.

4. Run Instructions

4.1. System Requirements

The program code is written in Python 3 and works in any operating system that supports the Python 3 language (e.g. Linux, OSX, Windows).

I have developed and tested the program in the following environment:

- MacBook Air, 1.7 GHz Intel Core i7, 8GB RAM
- Python 3.5
- Jupyter Notebook 4.1

Advice: You can download and install Python 3 and Jupyter Notebook together via the Anaconda package [13].

4.2. Starting the Program

After opening a Terminal (or Command line interface in MS Windows), unzip the file 'Code.zip', which has been included as part of the submission files for this project:

```
$ ls
Code.zip
$ unzip Code.zip
```

Code 4.1 Unzipping the Source Code Files with Source Data (from Terminal)

Next move into the directory 'code', which contains the Python source code and the initialisation file app.ini.

```
$ cd Code
$ ls
01-Image_cube.ipynb     app.py                  log
02-Image_dp.ipynb       dp.py                   support.py
03-Image_kmean.ipynb    images                  testdata
app.ini                 kmeans.py
$
```

Code 4.2 Moving into the Code Directory (from Terminal)

Now you can start the program. It is important that the program needs to be started from within the 'Code' directory.

```
$ python app.py
```

Code 4.3 Starting the GUI Application (from Terminal)

The following application window will appear:

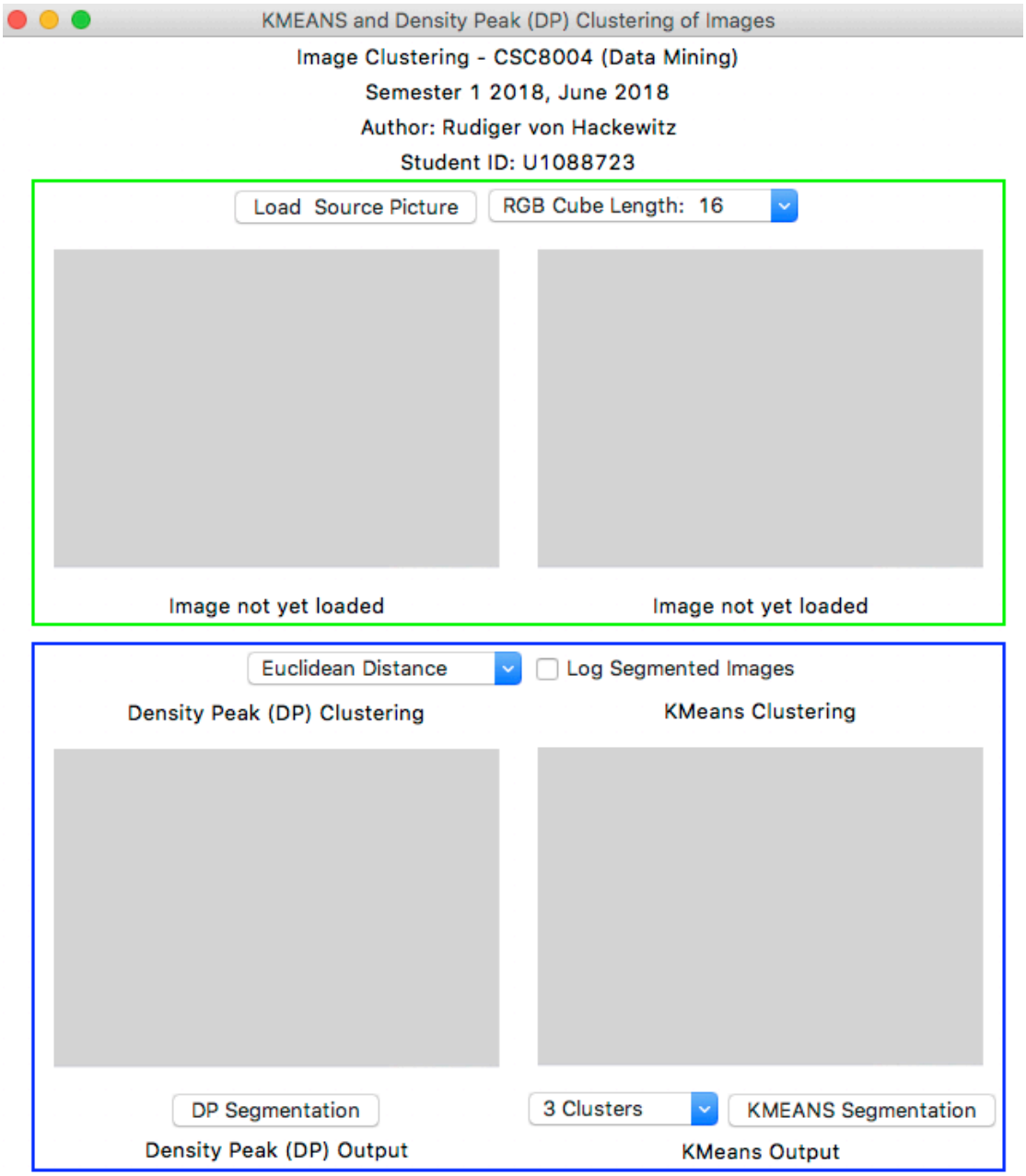


Figure 4.1 Entry Screen of the GUI Application

As no images are yet loaded, the screen is displayed with grey areas for the images.
The area framed with green colour contains the part of the program that is required for the pre-processing of images into mini-cubes. Button 'Load Source Picture' allows the user to load a source image for pre-processing.

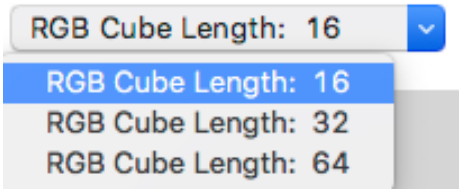


Figure 4.2 Dropdown List to Select Mini-Cube Size

Dropdown list 'RGB Cube Length' contains options to set the length of mini-cubes when slicing the data. The program has been optimised for the RGB Cube Length 16. Please do not change this default setting (unless you want to test against different hyper-parameters in Density Peak segmentation).

The area framed with blue colour contains the interfaces for running the DP and KMeans image segmentation processes.

4.3. Load Source Image

By pressing the 'Load Source Picture' button, a pop-up window appears that allows the user to select an image in sub-folder 'images'.
Filename 'emtpy.jpeg' is just used for the display of a grey image when the application is started and should not be selected as image:

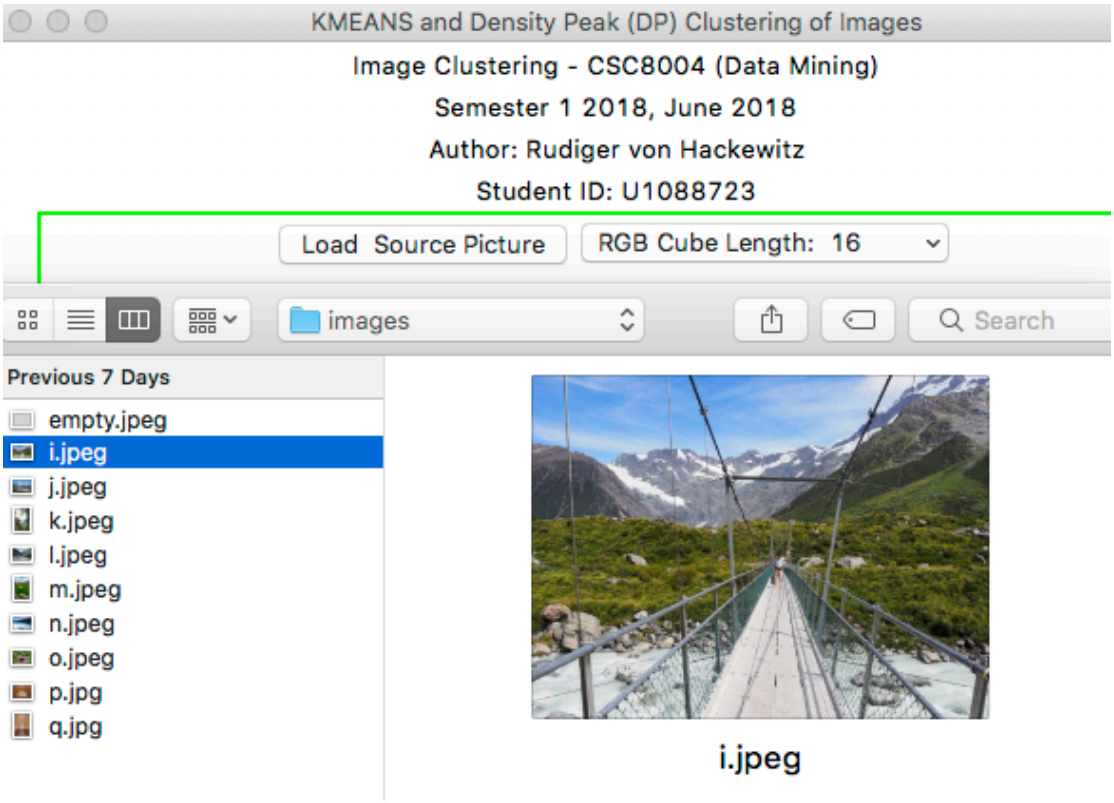


Figure 4.3 File Selection Box to Load Source Image

Once an image is selected, the original image and pre-clustered image are displayed in the area with the green frame:

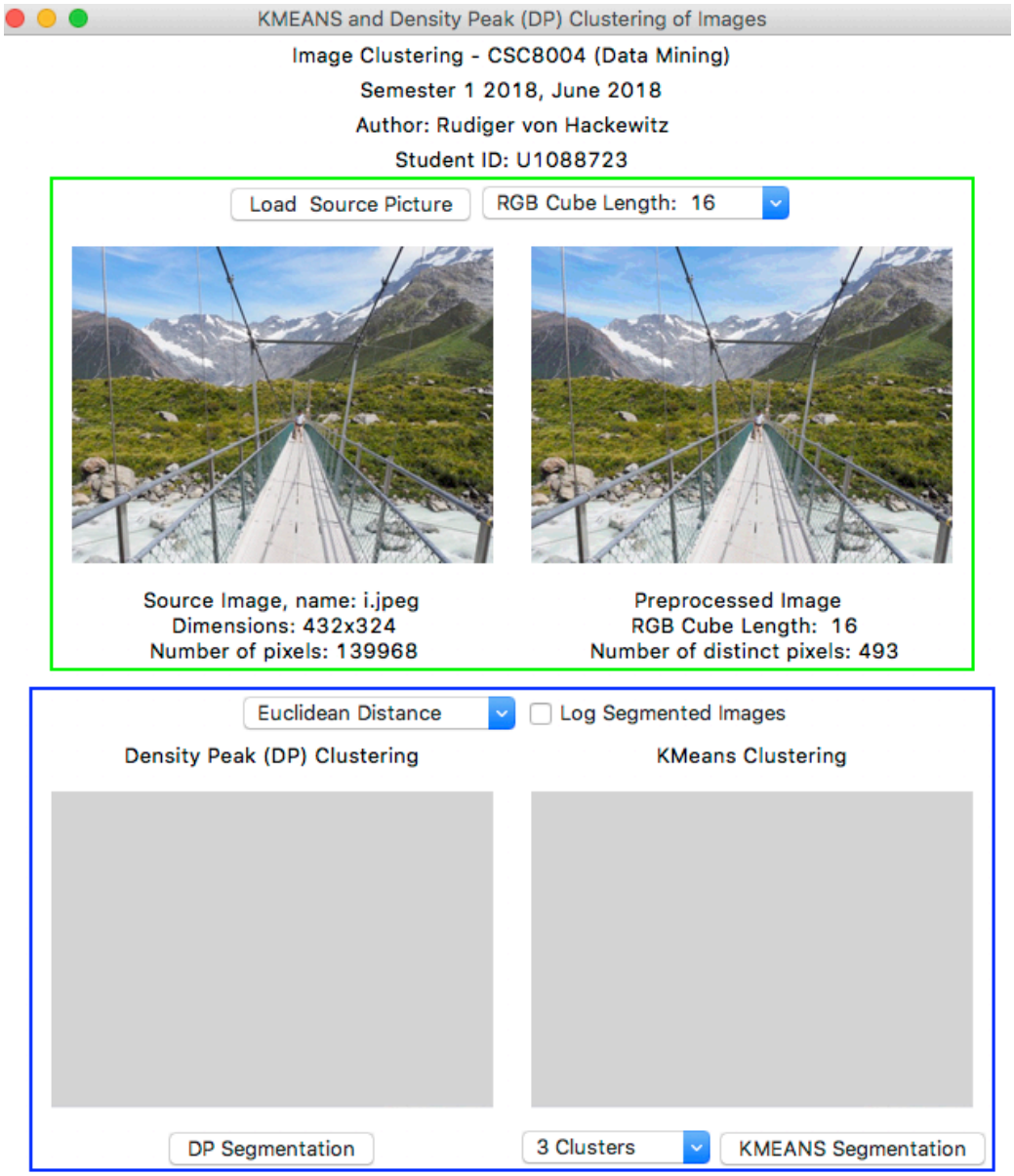


Figure 4.4 GUI Application after Loading the Source Image

The image to the left in the green frame displays the original image, with its file name, dimensions and number of pixels. The size information relates to the source file image, not the image that is visually displayed in the window.

The image to the right displays the pre-processed image (in this case with an RGB cube length of 16).

The number of distinct pixels (in the example 493) indicates the number of mini-cubes that contain image pixels. This number has been significantly reduced from 139,968 pixels in the original image, without any obvious loss of quality in the pre-processed image (at least for the human eye).

Now we are ready to run the DP and KMeans image segmentation processes on the pre-processed image.

4.4. Run DP and KMeans Image Clustering

In the section with the blue frame, we can control the runs for the DP and KMeans image segmentation processes.

4.4.1. Distance Metric

First we choose the distance metric for the segmentation process:

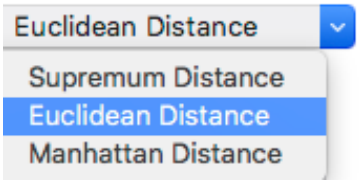


Figure 4.5 Dropdown List to Select Distance Metric

The user can select the Supremum, Euclidean or Manhattan Distance for running of the segmentation process. By default, the program sets the Euclidean Distance.

However, good cluster results can also be achieved with the Supremum and Manhattan Distance. The user is encouraged to compare segmentation results produced by all three metrics.

4.4.2. Running DP and KMeans Image Segmentation Processes

For the KMeans clustering process, the user needs to set an a-priori the number of clusters that should be used for segmentation of the image. The user has the option to choose between 1 and 10 clusters.

Pressing button 'DP Segmentation' and button 'KMEANS Segmentation' starts the DP and KMeans segmentation process, respectively. After completing the run, the window displays the two clustered images with their respective runtimes:

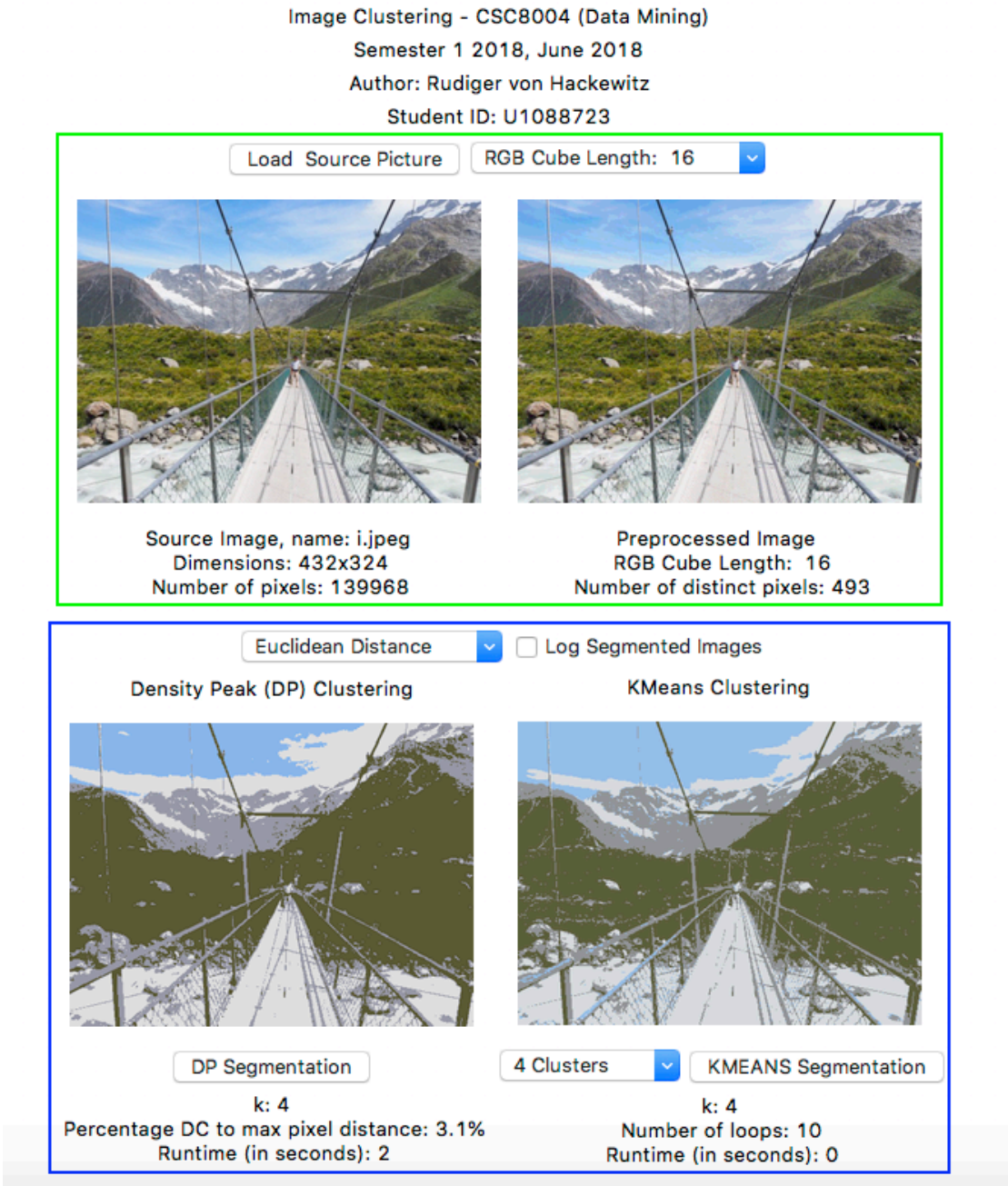


Figure 4.6 GUI Application after Running DP and KMeans Image Segmentation

For DP Segmentation, the process displays the percentage of the scaling factor DC to the maximum pixel distance in the image. The paper [1] by Zhensong Chen et al. recommends setting this value to 0.5%. However, in my project work I achieved best results with approx. 3%. The reason for the discrepancy in DC settings is explained at 3.1.2.

For the KMeans algorithm, the output includes the number of loops that had to be completed until the cluster centres stabilised and did not change any longer.

4.4.3. Logging Segmentation Results

When pressing the 'Log Segmented Images' checkbox in the blue image frame, the user can instruct the program to record segmented images in the sub-directory log/

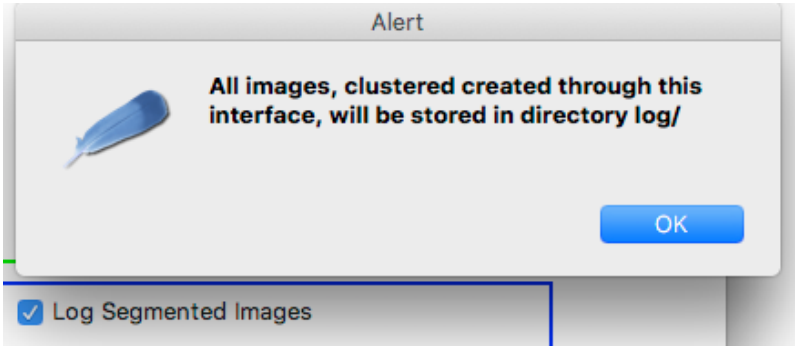


Figure 4.7 Warning Box when Ticking Checkbox 'Image Logging'

A popup window informs the user that image clusters are not just displayed in the screen, but also stored in the log folder. The clustered images are stored in the following format:

<source file name>_{ dp | kmean }_<k>_<RGB Length>_<distance metric>.jpeg

Code 4.4 Log Filename Format

The following screenshot from the Terminal lists example log files that have been created for DP and KMeans clustering:

```
$ ls log/
i.jpeg_dp_4_16_Euclidean.jpeg
i.jpeg_kmean_10_16_Euclidean.jpeg
i.jpeg_kmean_4_16_Euclidean.jpeg
$
```

Code 4.5 Sample of Logged Image Files (from Terminal)

In the output above, the original filename of the clustered images is i.jpeg; based on the filename info the DP algorithm has been run with k=4, RGB Mini-Cube Length = 16 and the Euclidean distance metric.
KMEANS clustering has been run with k=4 and k=10 against the Euclidean distance metric and the RGB mini-cube length 16.

4.5. Initialisation File app.ini

The initialisation file app.ini includes configuration and hyper-parameter settings that are critical for the operation of the program. It contains four sections and the parameters for each section are explained below.

Section	Parameter Name	Default Value	Description
APP			
	WindowSize	800x800	This defines the initial window size when starting up the application. Values should be increased (e.g. 1100x1100) when working on large desktop screens. Default settings are optimised for display on laptop screens with resolution of 1440x900.
	ImageHeight	200	The image height in the application window should be adjusted, based on the WindowSize. Default setting of 200 is optimised for display on laptop screens with resolution of 1440x900.
	ImageAtStartup	images/empty.jpeg	Grey image is displayed at start-up. This default setting can be changed.
	ImageSourcePath	Images/	This is the directory that contains all the source images. New images should be copied into this directory, if the program should run the image segmentation processes on those images.
	ImageLogPath	log/	If the log-tick box is ticked in the application window, then all clustered images are stored in this folder.
GLOBAL			
	Granularity	16	This setting defines the pixel length of the mini-cubes, when slicing the three-dimensional RGB image space. It can be a value in the geometric series of 2 (to ensure that all mini-cubes have the same length). Default setting of 16 should only be changed with care, as all the parameters for the DP clustering algorithm have been tuned for this value. Also – very fine settings for this parameter (e.g. 2,4 or 8) may cause very poor runtimes for the DP and KMeans clustering processes.
	DistanceMetric	Euclidean	Supported values are 'Supremum', 'Euclidean' and 'Manhattan'. The default value 'Euclidean' is displayed when starting the application. However, this value can be changed by the user in a drop-down list.
KMEANS			

	SetRandomSeed	yes	' SetRandomSeed' can be set to 'yes' or 'no'. If it is set to 'yes' then the KMEANS runs are reproducible, with the same Initialisation of centroids (and you can reproduce the clustered KMeans images in this report). It can be set to 'no' to ensure true randomness without seed (then two KMEANS runs with the same settings may produce different segmentation results)
DP			
	DensityScaling	400.0	This is a highly sensitive hyper-parameter in the DP algorithm. It should be adjusted with extreme care. It defines the scaling factor in the Gaussian kernel of the density model. Higher values increase the impact of pixels that are further away, and lower values will restrict the impact of the density value to pixels in close proximity.
	DecisionGraphScaling	100	Parameter defines the range of values in the Decision Graph; on the x-axis (density / rho) and y-axis (delta / distance). By default the range for both axes is set to 0 – 100. There should be no need to change this default setting.
	OutlierPercentage	0.20	This parameter is used for the outlier calculation of pixels in the Decision Graph. It is modelled based on the exponential distribution and sets the cut-off threshold for outliers on the distance axis (delta).
	DensityMin	0.05	If the density of a point falls below the threshold of DensityMin*DecisionGraphScaling, then it will no longer be considered as candidate for a cluster centre. The value for this parameter can be increased to control the granularity of clustered areas in a segmented image.

Table 4.1 Description of Parameter Settings in File app.ini

I tested and optimised the values in the initialisation file app.ini through Jupyter Notebook. The following section explains how to start Jupyter Notebook.

4.6. Using Jupyter Notebook (Optional)

Run the following command from the Terminal to start Jupyter Notebook[3]. The notebook needs to be started from within the 'code' folder, which does contain the .ipynb notebook files:

```
$ jupyter notebook
```

Code 4.6 Starting Jupyter Notebook (from Terminal)

This will start a user interface in a web browser window:

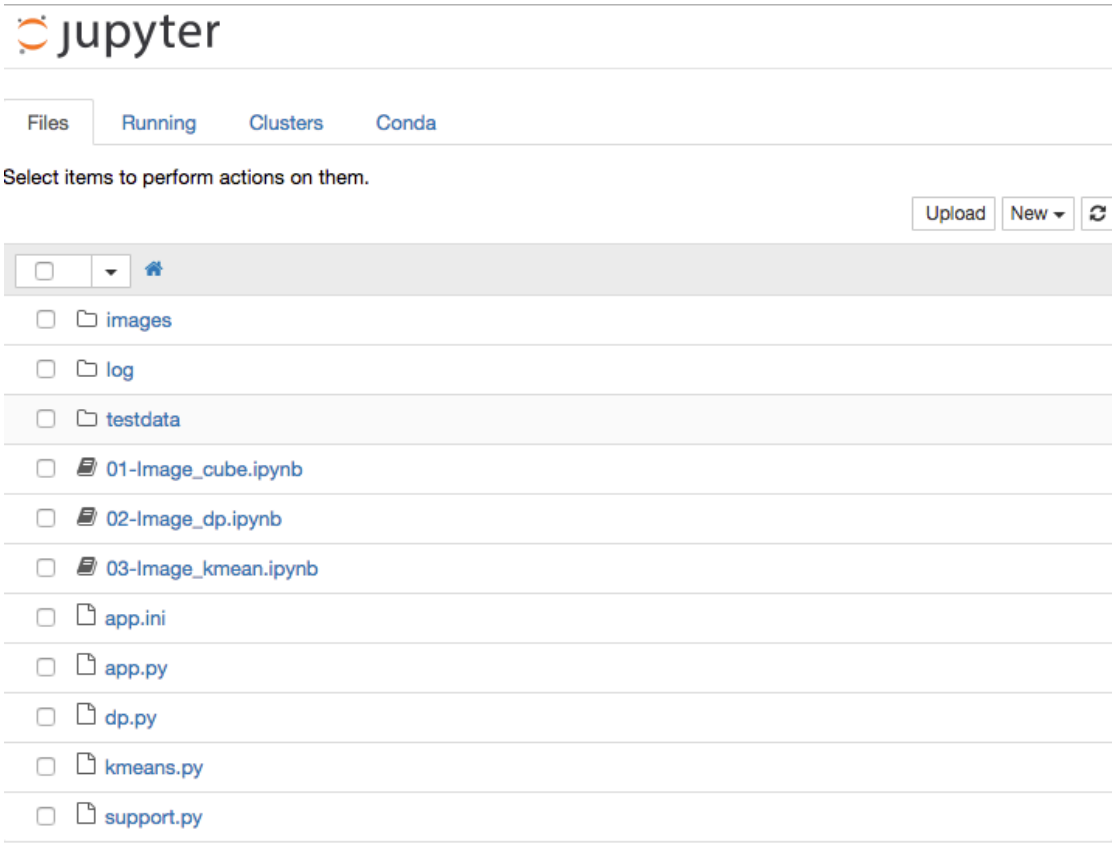


Figure 4.8 Jupyter Notebook Entry Screen

This interface allows you to view and edit the source code files (*.py) and the initialisation file app.ini. Most importantly, I have included 3 Jupyter Notebook files. They are not required to run the application. In fact – they can be deleted together with the subdirectory 'testdata'.
However, it is highly recommended to work through code and output in these workbooks, as it explains the journey that has been taken for development and testing of the code. Also, it includes some interesting graphs and findings from this report that cannot be reproduced through the GUI application.

Jupyter Notebook File	Content
01-Image_cube.ipynb	• Showcasing image qualities for various mini-cube settings • Simulating average distance between two pixels in a mini-cube (for Supremum, Euclidean and Manhattan metric)
02-Image_dp.ipynb	• Testing and illustrating DP clustering against manually created 20+ data points in 2-dimensional space • Testing of DP algorithm against images • Displaying data points in Decision Graph • Visualising outliers in Decision Graph
03-Image_kmean.ipynb	• Testing and illustrating KMeans algorithm against manually created 20+ data points in 2-dimensional space • Testing against Medical Data from Course CSC8003 in Semester 2 2017 • Testing KMeans algorithm against images

Table 4.2 Explanation of Jupyter Notebook Source Files *.ipynb

The notebook can be opened by simply clicking on the file name (e.g. 01-Image_cube.ipynb):

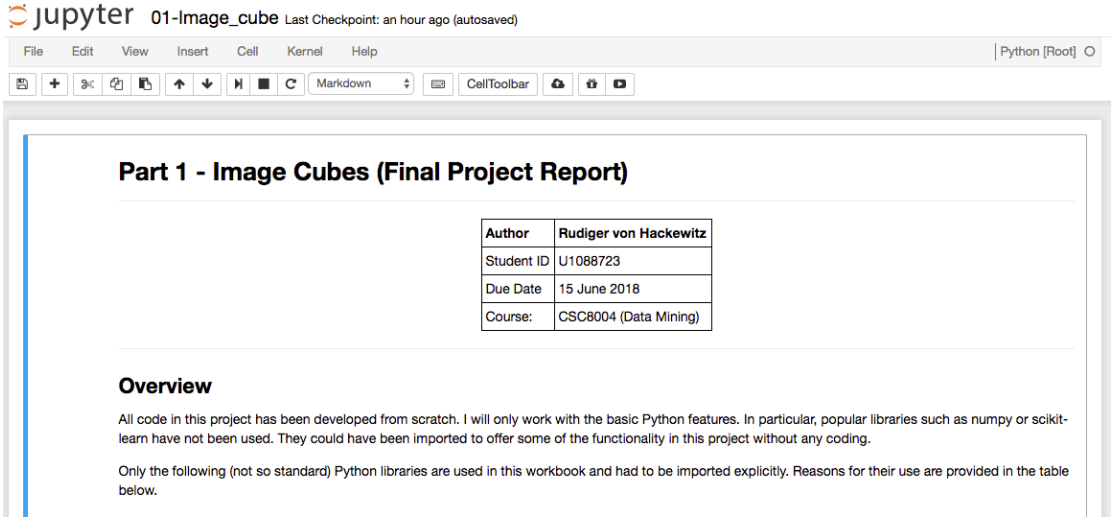


Figure 4.9 Jupyter Notebook File and Running the Cells

By pressing repeatedly the 'Run cell' button in the toolbar  , the user can work through code and text explanations in each cell of the workbook. To follow is a small extract of the workbook, which displays code and output for quality of pre-processed images with various granularity settings:

Figure 4.10 Jupyter Notebook Output (Example)

5. Test Results

I did not develop and test the KMeans clustering processes against images. This would have been very hard to unit-test and validate.

Instead, the two image segmentation processes were developed and tested against manually entered 20+ test data points in the Excel file testdata.xlsx. All test data for this project is provided in the sub directory testdata/

It is relatively easy to identify and fix bugs by testing the code on some 20+ two-dimensional data points. However, it is nearly impossible to undertake similar debugging activities on thousands of pixels in an image.

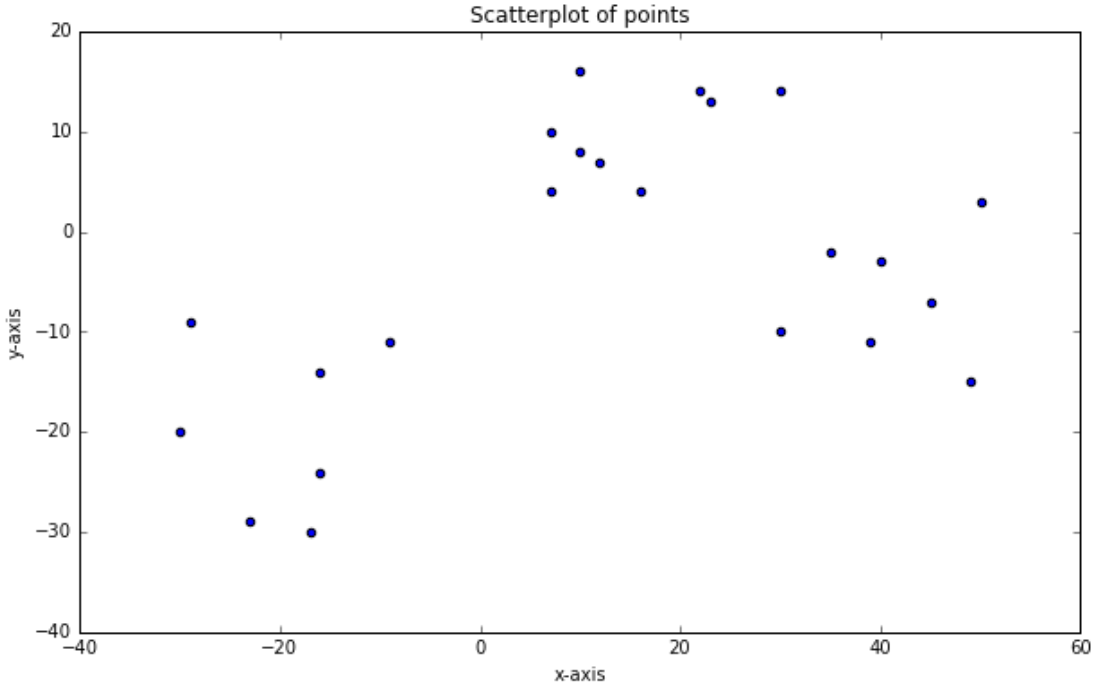


Figure 5.1 Scatterplot with Test Data for DP and KMeans Clustering

Intuitively we would group the two-dimensional test data points into three different groups: one to the top at the middle, a second to bottom at the left and the third to the right in the middle.

We test the newly developed clustering algorithms against this data and visualise the results, by displaying the points in different colours. As a starting point, I work with the Euclidean metric, as it is a plausible metric to describe the distances between points in this diagram.

5.1. Density Peak Clustering – Testing

Code and output for this section is provided in Jupyter Notebook file *02-Image_dp.ipynb*

We use the Gaussian kernel function to sum up calculated densities across all points and annotate each point with its calculated density. The formula for the calculation of the density is according to [1]

$$\rho_i = \sum_j \exp^{-\frac{d_{ij}^2}{d_c^2}}$$

Equation 6 Calculation of Density (Rho) in DP Algorithm

To follow is the code that implements Equation 6, and assigns each pixel its density rho. Please note that the average distance between two pixels is chosen, when they are located in the same mini-cube. The mean value for each distance measure has been simulated at 3.1.1.

```
# calculate the density for a given point in the list of points:
def density (self, p):
    # for points inside the same cube:
    # use the average distance between two random points in a cube.
    # The average distance depends on the distance metric:
    if self.dist == 'Euclidean':
        est = 0.66
    elif self.dist == 'Manhattan':
        est = 1.0
    else:
        est = 0.54  # Supremum distance
    est_dist = int (self.GRANULARITY * est)
    rho = (self.pnts[p] - 1) * exp(-pow(est_dist/self.dc,2))
    # now use the distance between centroids of each of the other cubes to add
    # to the calculation of the overall density for the point:
    for pp in self.pnts.keys():
        if pp != p:
            rho = rho + self.pnts[pp]*exp(-pow(dist_func(pp, p, self.dist)/self.dc,2))
    return rho
```

Code 5.1 Calculation of Density (Rho) for Each Pixel in DP Algorithm

The final densities for each of the points have been scaled in the range from 0 to 100 and displayed in the following scatterplot.

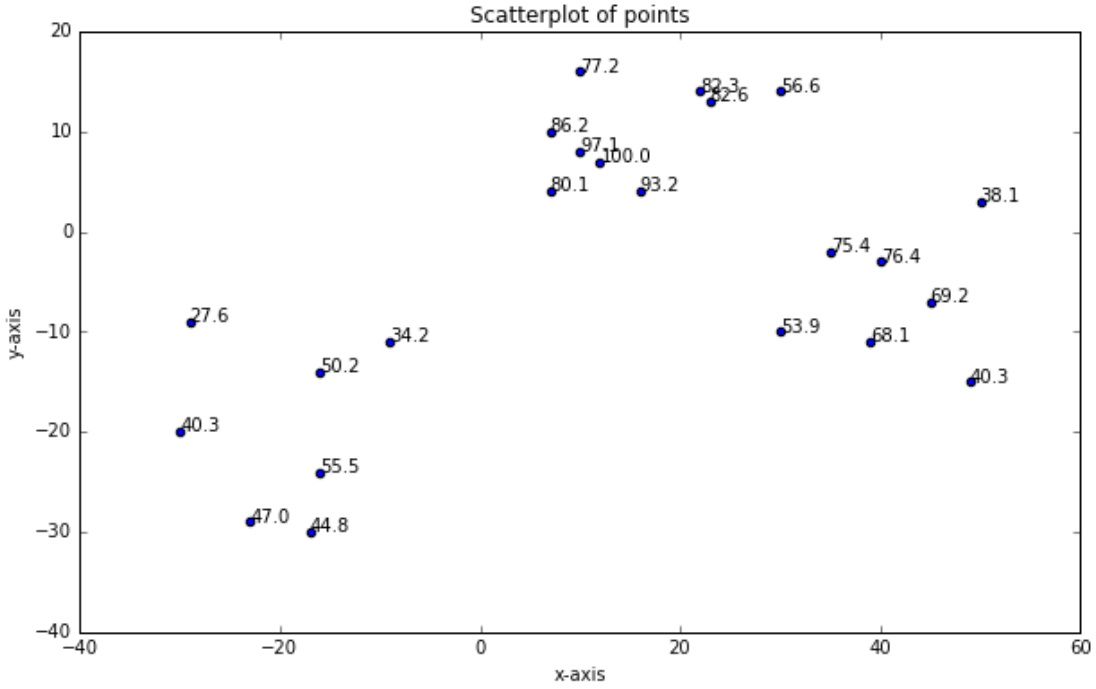


Figure 5.2 Scatterplot with Test Data and Assigned Densities (from 0 to 100)

The highest density (100) is for a point in the top cluster, then a high density point can be found in the cluster to the right and also in the cluster in the bottom left. Next we calculate a distance for each pixel, based on the closest point with higher density than the point itself. To follow is the formula for its calculation, based on [1]

$$\delta_i = \begin{cases} \min\limits_j(d_{ij}) & \rho_j > \rho_i \\ \max\limits_j(d_{ij}) & \rho_i \text{ is the highest density} \end{cases}$$

Equation 7 Calculation of Distance (Delta) in DP Algorithm

Equation 7 is implemented in the following code snippet.

```
# For each point calculate the minimum distance of the point to another point with higher or equal density:
def distance_points (self):
    pmax = max([p for p in self.pnts.keys()], key = lambda x : self.dens[x])
    self.dst[pmax] = max([dist_func(pmax, pp, self.dist) for pp in self.pnts.keys()])
    for p in self.pnts.keys():
        if p != pmax:
            self.dst[p] = min([dist_func(p, pp, self.dist) for pp in self.pnts.keys() if self.dens[pp] > self.dens[p]])
    # scale the distance value between 0 and DG_sCALING
    m = self.DG_SCALING/max(self.dst.values())
    for p in self.dst:
        self.dst[p] = self.dst[p]*m
    return self.dst
```

Code 5.2 Calculation of Distance (Delta) in DP Algorithm

In the next step, we convert the data to a Decision Graph, with density (rho) as x-axis and distance (delta) as y-axis.

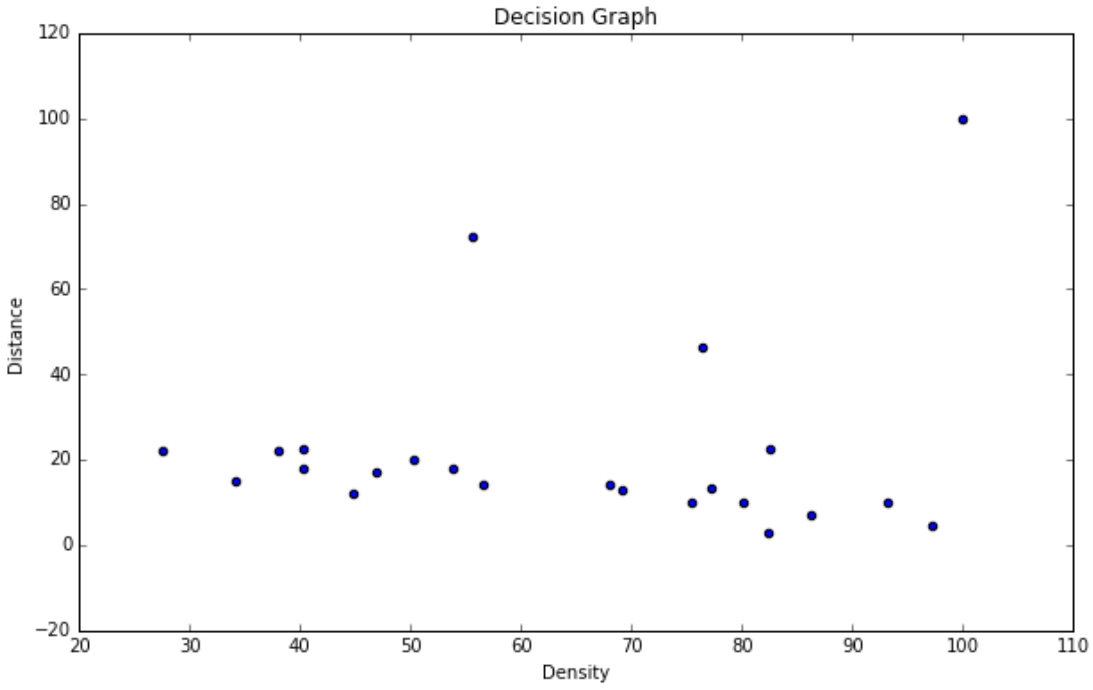


Figure 5.3 Decision Graph with Test Data

Three points can be identified as outliers. They are the designated three centroids of the clusters. Now we visualise the outliers in the Scatterplot.

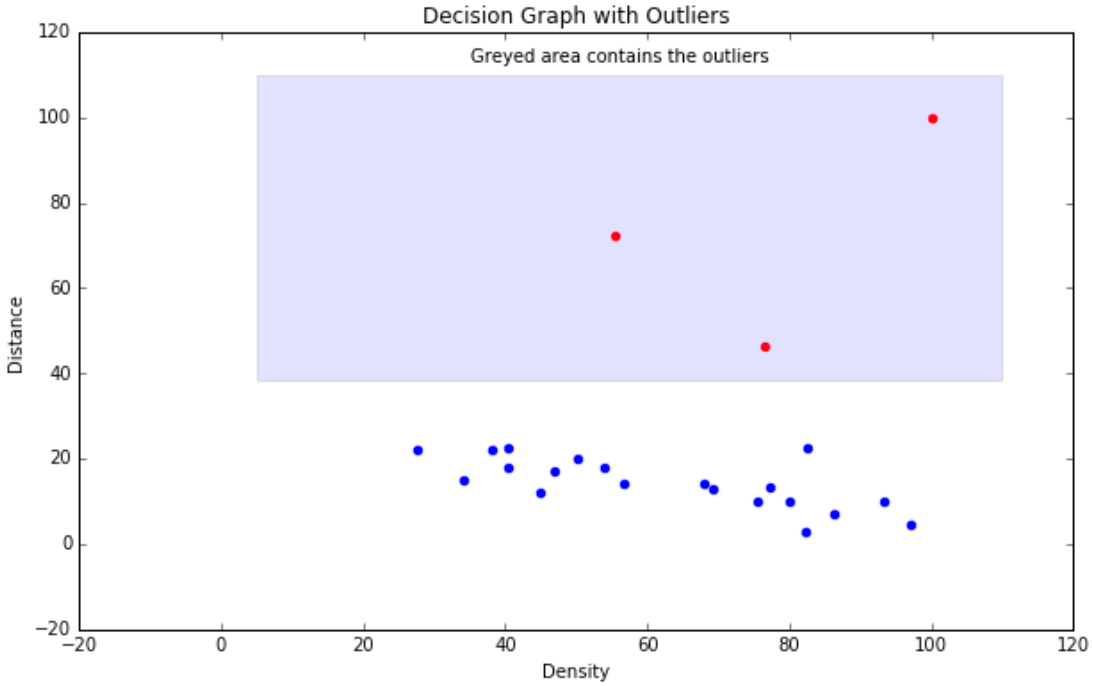


Figure 5.4 Highlighting Outliers in the Decision Graph with Test Data

Points in the greyed area are classified as outliers and marked as the cluster centres. We take as minimum value for the density 5% of the density and as minimum value for the distance a value that is modelled by the exponential distribution. The resultant outlier area is shaded in grey.

Finally, we use those outliers and assign the remaining points to their appropriate groups. According to [1], a point x_i is assigned to the same group as x_j, if it meets the following two conditions:

$$(1)\ \rho_j > \rho_i$$

$$(2)\ d_{ij} = \min_{l \neq i}(d_{il})$$

Equation 8 Assigning Remaining Pixels to Segments in DP Algorithm

A recursive algorithm, as described in 3.1.4, has been developed to assign the remaining points.

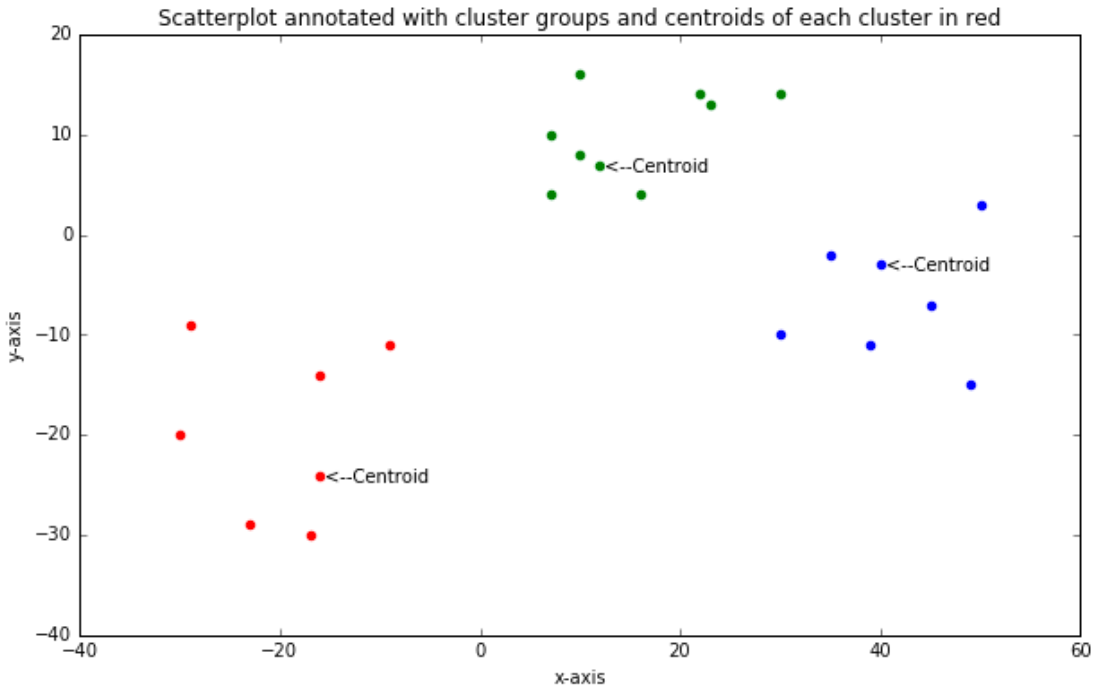


Figure 5.5 Final DP Cluster Result with Highlighted Cluster Centres

The three groups in the final scatterplot are coloured and produce intuitively correct results as they had been predicted when the raw data had been displayed in Figure 5.1. The centroids are somewhat in the centres of their respective groups.

Successful completion of the DP clustering algorithm against this (small) test data set gave me confidence that its implementation is sound and should also work correctly when it is applied against larger data sets such as pixels in images.

5.2. KMeans Clustering – Testing

All results in this section are coded and taken from the Jupyter Notebook file *03-Image_kmean.ipynb*. This file is attached to the submission and the user is encouraged to re-run the code in this workbook.

5.2.1. Test Data Set

I worked with the same test data set, as provided in the Excel file, and applied a KMeans clustering algorithm with 3 clusters (k=3).

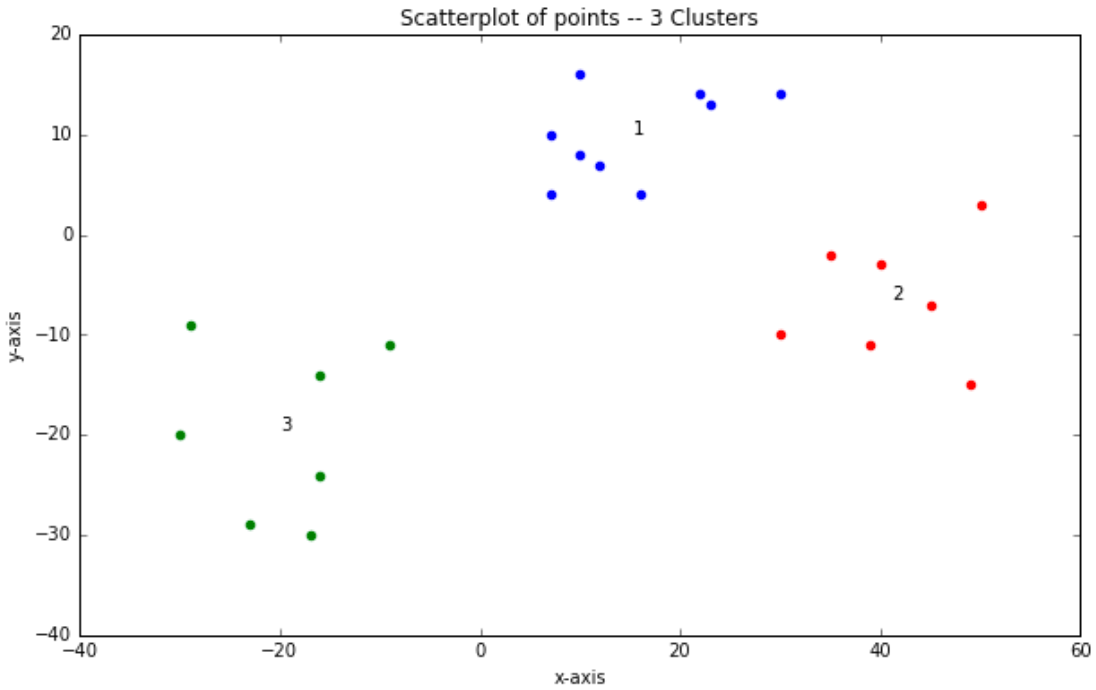


Figure 5.6 Clustered Test Data after Running the KMeans Algorithm

I have visualised the centres of each cluster with its number. E.g. group 1 with the blue dots is at the top of the diagram, with the 1 nicely positioned as the centre of the points. The same can be observed for the groups 2 (red points) and 3 (green points).

This confirms that the developed KMeans algorithm produces reasonable results.

5.2.2. Medical Data

Next I test the developed code in this class against previous work from S2 in 2017, course CSC8003 'Machine Learning', assignment 2, 26 September 2017. In this previous assignment, under heading '5.1 K-Means', I have applied KMeans clustering, using core functionality of the R language [6].
The purpose of the clustering was to distinguish between the two different groups of awake and asleep patients, based on human brain activity, as measured by an Electroencephalograph (EEC). The normalised data from the EEC - as they have been used in the aforementioned assignment - are provided in an Excel file and available in the testdata sub-directory.

I applied the Python code against the same data from this Excel file, and compared the results with those of the R-code in the previous assignment.

Cluster centres as calculated in last year's project work:

k-means clustering

```
# set seed function (to make it reproducible)
set.seed(123)
# pick columns x3, x5 and x6 only; create two clusters
kmean_cluster <- kmeans(x <- dfst_data[, selected_features], centers <- 2)
kmean_cluster$centers
```

```
##          x3         x5         x6
## 1  0.588163 -0.5478847 -0.5516517
## 2 -1.313226  1.2232944  1.2317050
```

```
table(kmean_cluster$cluster)
```

```
##
##    1    2
## 1036  464
```

The K-Means algorithm assigns 1036 records to 'Group 1' and 464 records to 'Group 2'.

Cluster centres as calculated in this year's project work:

Group	x3	x5	x6
Group 1	0.5881629545741901	-0.5478847378232247	-0.5516516529578904
Group 2	-1.3132259071958523	1.223294371519095	1.2317049837594267

It is remarkable that the cluster centres for the two groups (awake / asleep patients) are identical, up to rounding differences!

Also - number of records in both groups are identical:

Group 1	Group 2
1036	464

Figure 5.7 Comparing KMeans Result with Centroids Produced by R

This new KMeans algorithm and the implementation in the R package produce identical results. This gave me confidence that the implementation of the KMeans clustering algorithm is sound and can be tested against images.

6. Evaluation

In this section I highlight some of the findings when I applied the DP and KMeans segmentations on pictures, taken during my holidays in January 2018 in New Zealand.

The images referred to are included in the submission, in folder images/. All pictures have been pre-clustered with the mini-cube length of 16, and KMeans has been executed with a seed value (set in the app.ini file). The reader can reproduce the DP / KMeans clustered images below via the GUI app.py.

6.1. Detection of Fluid Changes in Colour Schemes

The first observation is related to the pre-processing of images in mini-cubes, effectively reducing the number of available colours (and distinct pixels) in an image:

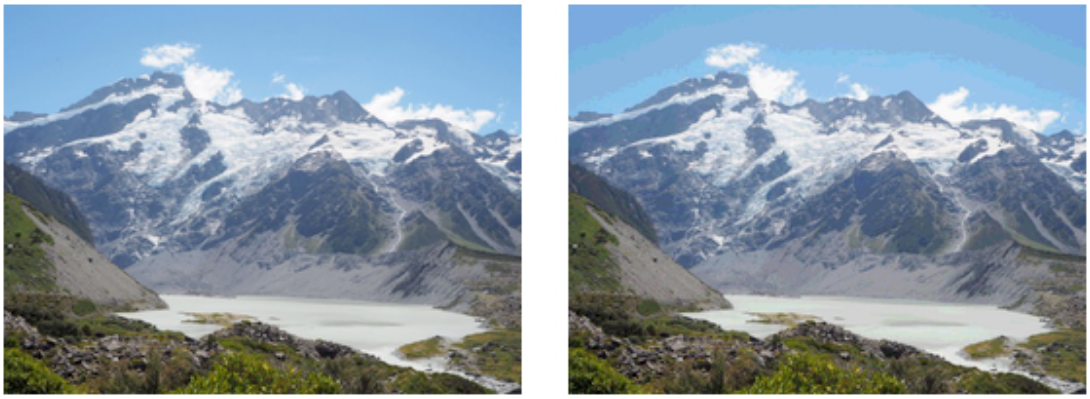


Figure 6.1 The Pre-Processed Image Shows a Halo above the Mountain Range

The change of colour in the sky in the original image on the left is hardly detectable for the human eye. However, in the pre-processed image to the right, it is clearly visible that the sky is lighter close to the mountains and darker further away. Some sort of 'halo' appears across the mountain range in the pre-processed image.

6.2. Distance Metrics

I tested my holiday photos against the Supremum, Euclidean and Manhattan distance metrics. To follow I depict the results for one of the pictures, that was taken while racing in a speedboat across Lake Wakatipu near Queenstown:

Source Image, name: n.jpeg
Dimensions: 576x432
Number of pixels: 248832

DP Clustering – Supremum Distance

DP Segmentation

k: 4
Percentage DC to max pixel distance: 5.4%
Runtime (in seconds): 1

DP Clustering – Euclidean Distance

DP Segmentation

k: 4
Percentage DC to max pixel distance: 3.2%
Runtime (in seconds): 1

DP Clustering – Manhattan Distance

DP Segmentation

k: 4
Percentage DC to max pixel distance: 2.0%
Runtime (in seconds): 1

Figure 6.2 Clusters Produced by Supremum, Euclidean & Manhattan Metric

It is very hard to pick a metric that clusters 'best'. However, in general it could be observed that the Euclidean distance adds more detail to the clusters, whereas the Supremum distance tends to produce larger areas with less detail.
It will depend on the nature of the underlying problem, which metric should be used.

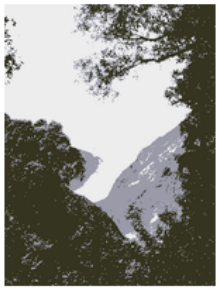
6.3. DP vs KMeans Clustering

In the following picture series, I have run the KMeans algorithm with the same value for k, as it had been determined by the DP cluster process for a source image. This allowed me to compare the segmentation results of the two segmentation processes:

Source Image, name: k.jpeg
Dimensions: 324x432
Number of pixels: 139968

Euclidean Distance | Log Segmented Images

Density Peak (DP) Clustering



DP Segmentation

k: 3
Percentage DC to max pixel distance: 3.1%
Runtime (in seconds): 4

KMeans Clustering

3 Clusters | KMEANS Segmentation

k: 3
Number of loops: 7
Runtime (in seconds): 0

Source Image, name: l.jpeg
Dimensions: 576x432
Number of pixels: 248832

Euclidean Distance | Log Segmented Images

Density Peak (DP) Clustering

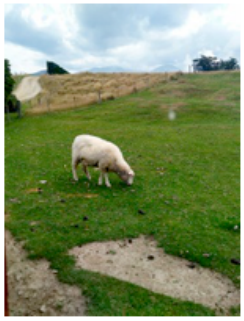
DP Segmentation

k: 3
Percentage DC to max pixel distance: 3.2%
Runtime (in seconds): 1

KMeans Clustering

3 Clusters | KMEANS Segmentation

k: 3
Number of loops: 3
Runtime (in seconds): 0



Source Image, name: m.jpeg
Dimensions: 180x240
Number of pixels: 43200

Euclidean Distance | Log Segmented Images

Density Peak (DP) Clustering

DP Segmentation

k: 3
Percentage DC to max pixel distance: 2.8%
Runtime (in seconds): 1

KMeans Clustering

3 Clusters | KMEANS Segmentation

k: 3
Number of loops: 10
Runtime (in seconds): 0

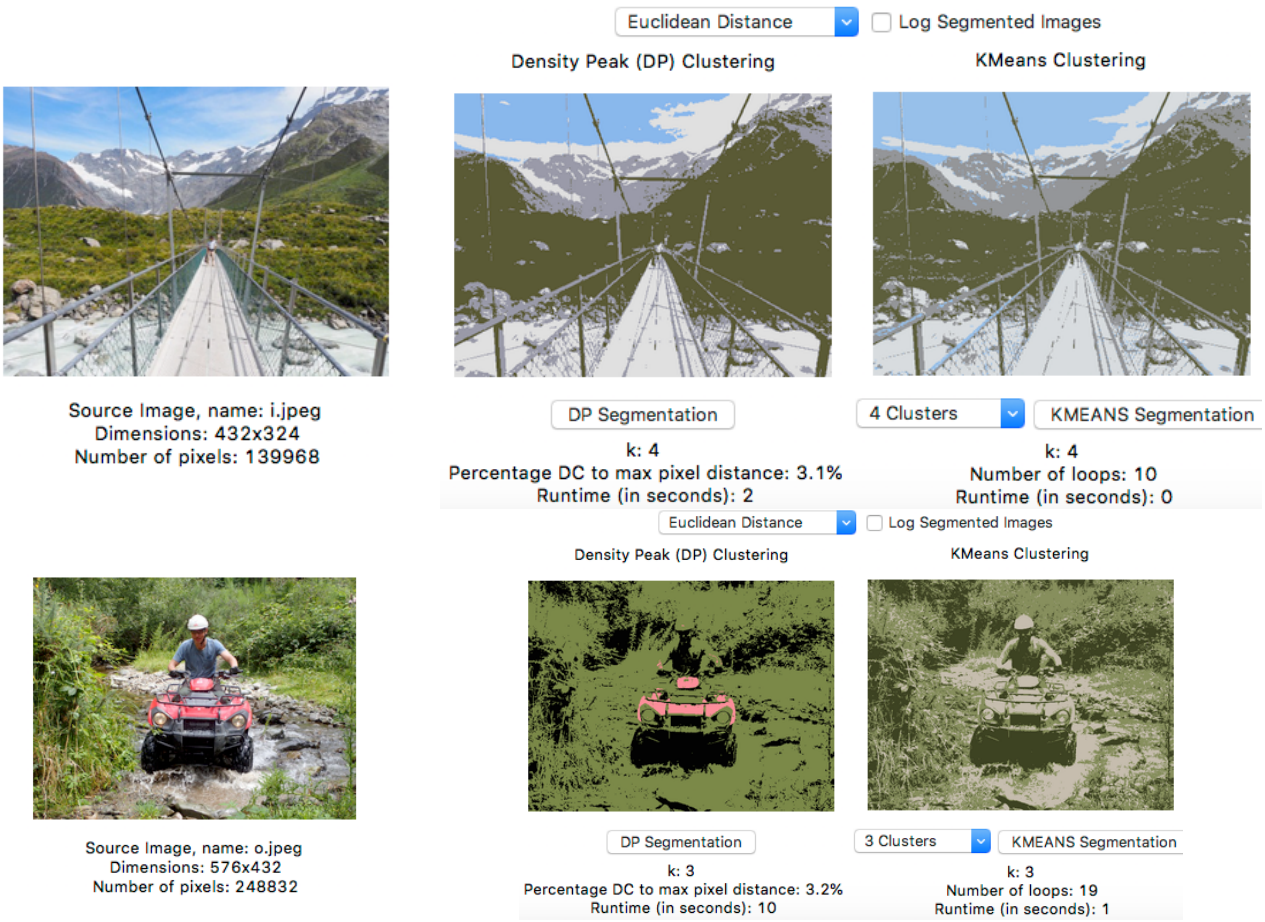


Figure 6.3 Comparison DP vs KMeans Image Segmentation

It is an interesting observation that the KMeans algorithm converges very quickly. In all the test runs above, KMeans cluster centres stabilise after less than 20 iterations (loops)! As a result, the KMeans algorithm runs generally faster than DP clustering. See also the slightly higher runtimes in the outputs for the DP algorithm above.

In general, DP tends to produce pictures with starker contrasts. KMeans produces more balanced colour schemes for the human eye. The differences in the cluster regions for the two processes appear to be marginal, when comparing the results.

The only significant difference in colour can be found in the last source picture with the quad-bike. The red frame of the quad-bike does only appear in the DP clustering algorithm. KMeans 'evens out' relatively small colour areas in a picture. This phenomenon is further investigated in the following section.

6.4. Outlier Detection in DP Algorithm

Outlier detection of pixels in the Decision Graph of DP and tuning of outlier detection has been undertaken in the Jupyter Notebook file *02-Image_dp.ipynb*. The clustered image for the quad-bike offers an excellent example:

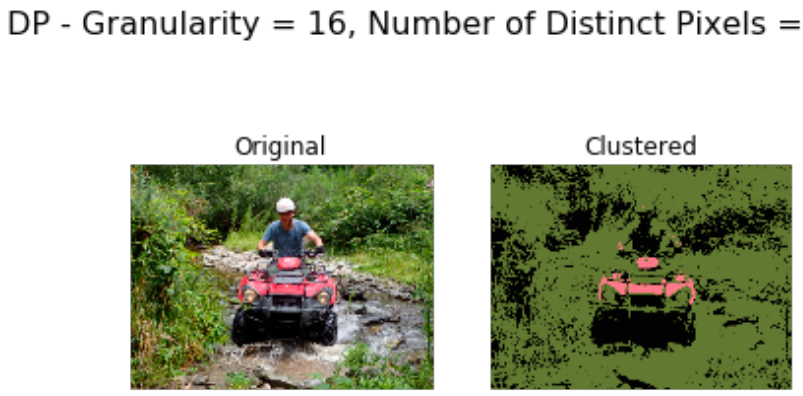


Figure 6.4 DP Clustered Quad-Bike Image (Default Parameters)

For this picture, the DP algorithm identified the following three outlier pixels in the grey-shaded rectangle of the Decision Graph:

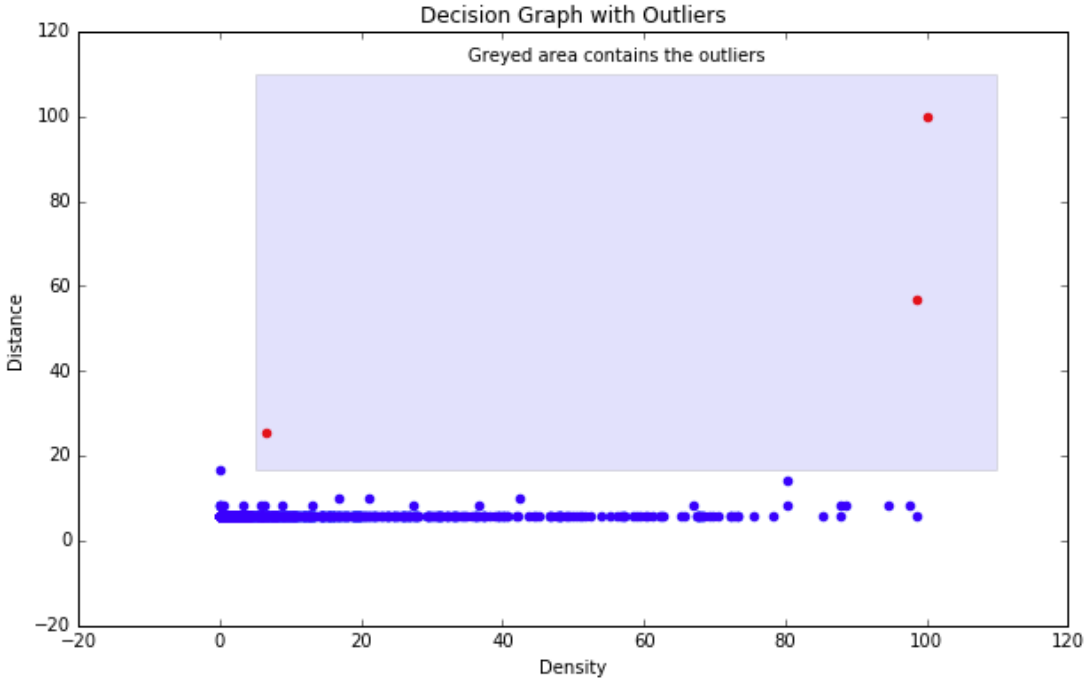


Figure 6.5 Decision Graph for Quad-Bike Image (Default Parameters)

The pixel for the red colour in the quad-bike is in the bottom left corner: it has relatively low density, because red covers only a very small portion of the entire photograph.

We can exclude this red area in the cluster process by simply increasing the hyper-parameter *DensityMin* in the app.ini file from 0.05 to 0.10 and then re-run the process:

DP - Granularity = 16, Number of Distinct Pixels = 1122

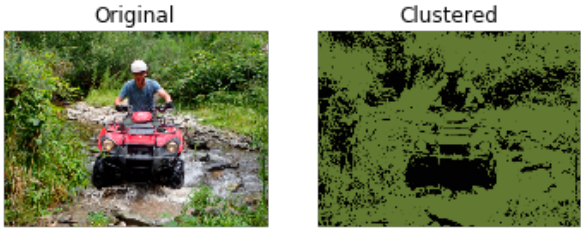


Figure 6.6 DP Clustered Quad-Bike Image (Increased Density Threshold)

Now the red spot in the clustered image disappears, as the red pixel is now no longer considered an outlier in the Decision Graph below:

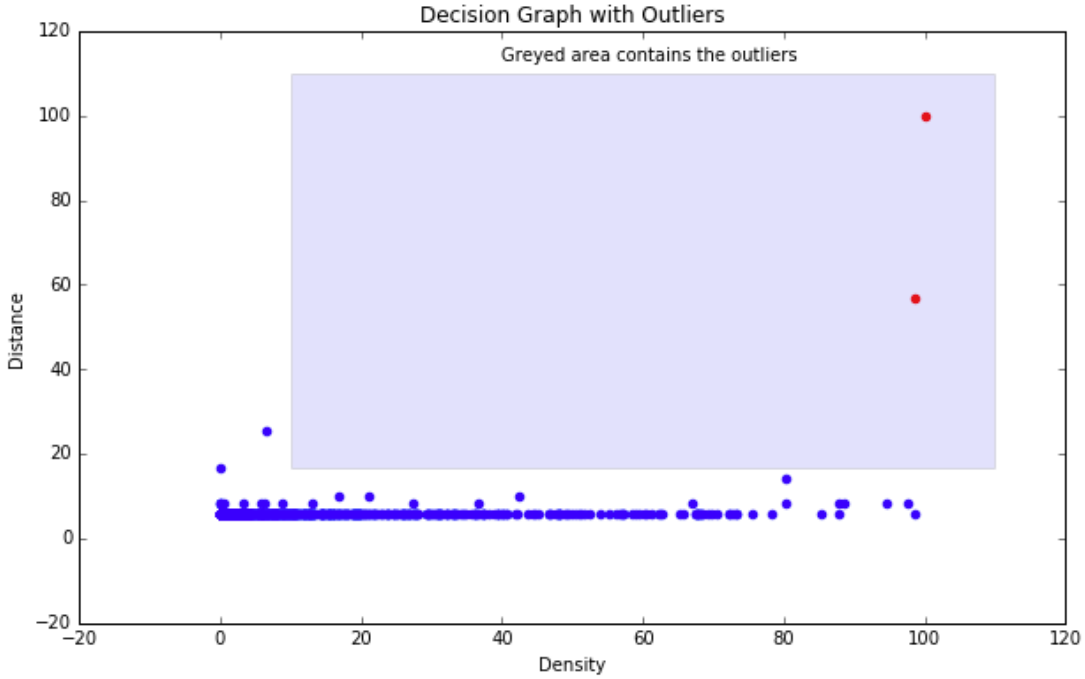


Figure 6.7 Decision Graph for Clustered Quad-Bike Image (Increased Density Threshold)

The pixel to the left (representing the colour red in the quad-bike) is now outside the area that is picked as the cluster centres of the picture.

However, we could also decide to include the pixel at the bottom as outlier, as it just misses out and is below the distance threshold. After setting *OutlierPercentage* to 0.30 in app.ini and *DensityMin* back to 0.05, we re-run the DP process again and now we get:

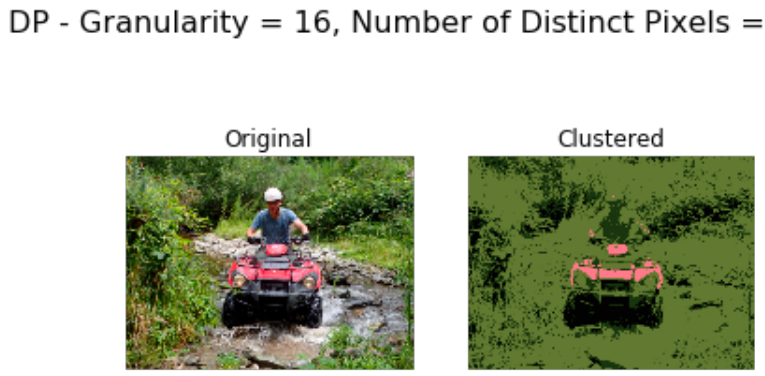


Figure 6.8 DP Clustered Quad-Bike Image (Decreased Distance Threshold)

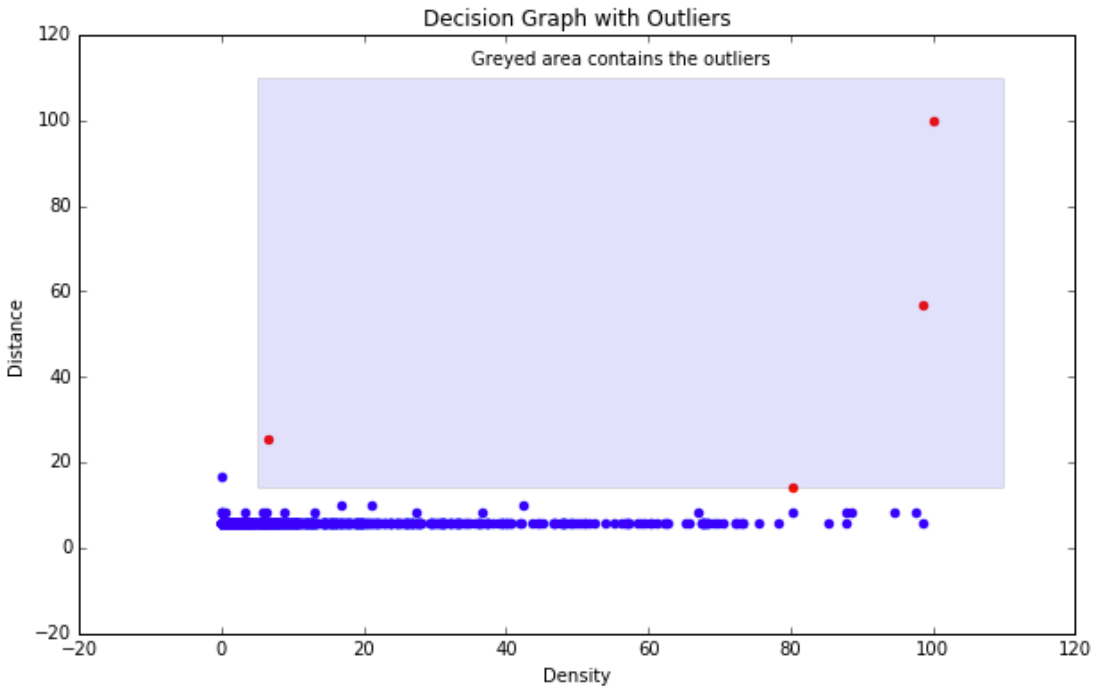


Figure 6.9 Decision Graph for Clustered Quad-Bike Image (Decreased Distance Threshold)

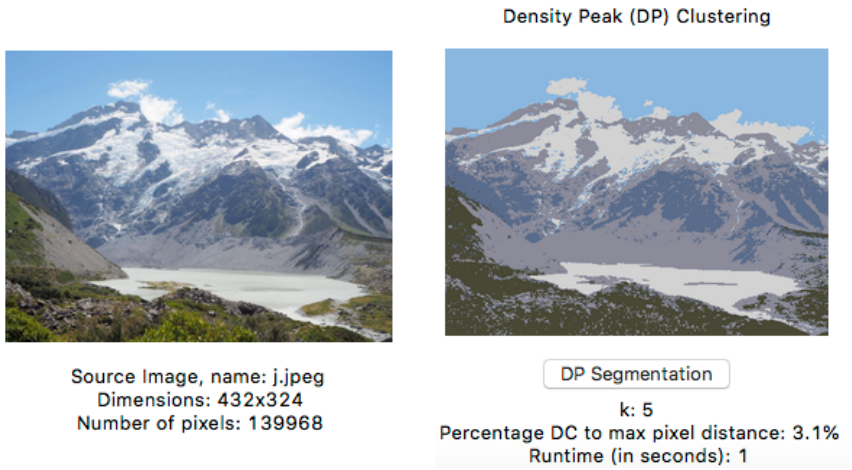
Now we end up with 4 cluster centres, as the point to the bottom is now included as cluster centre.

This illustrates how the hyper-parameter settings for the DP algorithm can be modified to work towards specific user preferences when clustering the picture.

6.5. Randomness of KMeans Clustering

After setting *SetRandomSeed = no* in the app.ini file we can run the KMeans clustering process repeatedly for the same image with different random initialisation clusters to start with. It then produces some image clusters that are surprisingly different to the others! To follow is an example.

We start loading the following source image and run DP clustering:



DP segmentation produces a reasonable result with five image clusters (k=5). Now we set k to '5 Clusters' for the KMeans process and run KMeans multiple times. To follow are three of the produced cluster images:

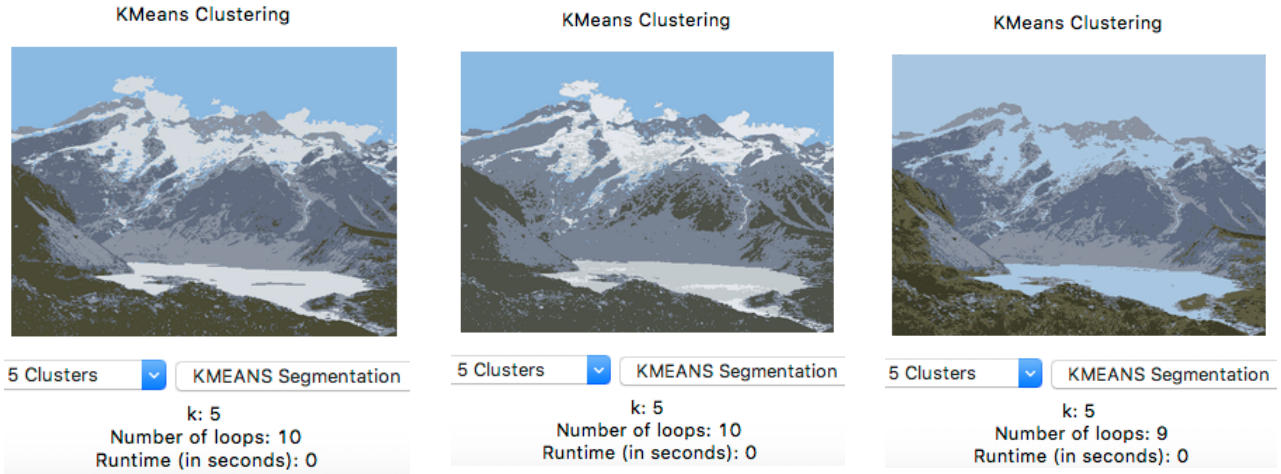


Figure 6.10 KMeans Clustered Image of Mount Aoraki with three Different Initialisation Centres

KMeans produces some surprisingly different image clusters for different initial centroid settings. For example, in the image to the right there are no clouds at all in the segmented image!

In contrast, DP produces a deterministic clustered picture.

6.6. KMeans vs DP – The Verdict

In my project proposal I set out to compare the clustering results of the KMeans and DP algorithms. It turned out that both algorithms produce similar clustering results against the test images (when using the same number k of image segments in the two algorithms). To follow are the advantages of each approach.

DP – Pros:

- DP was better at identifying some unusual outliers in colour patterns (such as the colour red in the picture with the quad-bike)
- K is not an input parameter in DP clustering. This is helpful in situations where a program needs to process hundreds of images in bulk and automatically set an optimised cluster number k for each picture as part of the process.
- DP, contrary to KMeans, produces deterministic clustering results. A second run with the same hyper-parameter settings will always produce the same image segmentation.
- By adjusting the hyper-parameter settings in the DP algorithm, the user has some control over the segmentation process, e.g. by adjusting the process for outlier detection or the density-scaling factor DC. In contrast, the only input parameter for the KMeans clustering process is the number of clusters k.

KMeans Pros

- Overall, KMeans produced more balanced colour schemes that were somehow easier for the human eye to interpret.
- Should the user want to experiment with different settings of K in the same image, and then pick the 'optimal' number of clusters in an image, then KMeans is a good option.
- Runtimes of KMeans were generally better than of DP; this could be an argument to work with KMeans when runtime performance is critical.
- KMeans can run and complete clustering of images on the raw pixels of an image (though it can be very slow). In contrast, DP could not operate and complete the process on the raw pixel data in reasonable time, and pre-processing of images is a necessity in DP segmentation.

Final verdict: there is no clear 'winner', and it depends on the situation which algorithm is preferable.

7. Lessons

This project taught me several valuable lessons; some of them I want to briefly share in this section.

Firstly, image segmentation can be useful to test the performance of generic clustering processes. It might be difficult to get hold of large volumes of real data (e.g. blog posts or other personal data). But it is easy to use photographs with 100,000+ pixels and test prototypes of clustering algorithms against the pixels in those images. The finally developed code might then be used in a completely different domain (e.g. sentiment analysis in blog posts or others).
In many cases it is easy to build (or just copy & paste) clustering algorithms that will work well on smaller data sets. For example, there are many basic KMeans implementations with code-snippets available on the Internet. However, when developing code for large data sets in the real world, it is important to test against 'big data'. This is often forgotten when proto-typing and testing a data-mining algorithm.
In my initial project proposal, I set out to implement two different clustering algorithms (KMeans and DP). However, in reality I ended up building a third one (with mini-cubes for the pre-clustering of images). Sometimes, data in their raw format are not suitable for large data processing activities, and some creativity might be required to convert the data into a format that presents itself better for streamlined processing by a data-mining algorithm.
I anticipated that KMeans would perform poorly on very large data sets (e.g. image pixels with 100,000+ data points). To my surprise, the algorithm converged relatively quickly with all random cluster centre initialisations. Most KMeans runs could complete in less than 10 iterations! This explains why KMeans offers better runtime performance than DP; the later has to complete more complex calculations of the order $O(n^2)$.

8. Outlook

Further work is required to refine the calculation of the DC hyper parameter in the DP algorithm. In particular, it would be useful to develop a formula that provides meaningful clustering for a range of different RGB mini-cube side lengths. In summary, this would include the development of more generic algorithms to determine outliers in a Decision Graph.

I did not succeed in constructing well-performing DP and KMeans algorithms on the pixels in the source image, without any pre-clustering. This might be something worthwhile to investigate. It would also allow me to provide a more useful comparison with the results in the paper [1] by Zhensong Chen et al.

In my literature review, the use of the Euclidean metric was assumed as 'a given' by most authors. However, considering the computationally less complex calculations of the Manhattan and Supremum metrics, they could be viable alternatives; their use in image clustering should be further investigated, as they also produced excellent clustering results.

9. Literature Review

Image segmentation is used across a wide field of disciplines, such as object recognition, face extraction or medical imaging for diagnosis and detection of diseases.

Over the last decade, many research articles have been published about image segmentation; this is still a very active field of research, and various variants of K-Means clustering form the basis for many recently published image segmentation processes.

One of the challenges of the KMeans clustering algorithm is its unpredictable performance: based on the initial randomly assigned cluster centres, the number of iterations, and subsequently its runtime and the final segmented image result, may vary significantly. Yousef Farhang tackles this problem in his paper [7] by proposing an improved K-Means clustering algorithm for face extraction from images. His paper also highlights that clustering algorithms should be developed and tested against specific problems (e.g. face recognition) to yield better results.

Fuzzy C-Means (FCM) clustering is another popular image segmentation process that works similar to the standard K-Means process: rather than assigning image pixels to one of the K groups, FCM assigns probabilities to the pixels, how likely it is that they belong to a certain group. This probability is then revised in iterations, until a defined loss-function reaches a minimum. Keh-Shih Chuang *et al*. improve the FCM algorithm in their paper [9], by incorporating spatial information into the loss function. This revised FCM algorithm is less sensitive to noise in image pixels and produces 'smoother' segmentation areas.

Laurent Condat proposes a 'convex' K-Means approach for image segmentation [11]. This variant of K-Means uses spatial information in the image pixels and is based on the prior discretisation of the image into several areas. Centroids are then assigned to those areas, and simplify the process of finding a segmentation by using a 'convex' loss function.

Recent research papers have been released covering the topic of image segmentation for automated analysis of medical images:

Takayasu Moriya *et al*. explore in their paper [10] improvements to unsupervised image clustering for the diagnosis of lung cancer. They propose the image processing in two phases: first spherical K-means are applied to learn the centroids of X ray images with lung cancer. In a second step, the traditional K-means clustering is applied on pre-processed images. They highlight that their approach produces overall better results than traditional clustering methods.

H.P. Ng *et al*. discuss in their paper [8] the combination of the K-Means algorithm with the conventional watershed algorithm to medical images. The K-Means algorithm helps prevent the over-segmentation of medical images into too many regions, which is common in the traditional watershed algorithm.

Piyush M. Patel proposes a revised K-Means algorithm for the detection of brain tumors. He tackles the issue that different initial settings for the K centroids may produce different final centroids (and segmentation results). He proposes to base the final segmentation on a series of initial samples for the K cluster centres, and choose the final cluster centres, based on a minimised square-error function.

Many of the recent image segmentation algorithms use some elements of traditional K-Means clustering. Recent project work does not just focus on pixel colours for clustering, but also spatial elements (proximity and density of similar pixels to each other). Rather than developing generic image clustering algorithms, there is a trend to model and test image segmentation processes against specific problems, such as in the medical field (e.g. diagnosis of cancer) or biometric recognition (e.g. faces, the iris of an eye or a fingerprint).

10. Source Code Authorship

I have developed all source code in the project from scratch. This includes the code for the development and testing of the KMeans and DP segmentation algorithms with pre-processing of images (in source code files kmeans.py and dp.py respectively), supporting functions (support.py) and the code for the GUI interface in the Python source code file app.py.

I developed and included complementary code in the Jupyter notebooks so that the interested reader can get a better understanding about the intrinsic mechanisms of the DP and KMeans clustering algorithms.

In particular, the notebooks illustrate how I came to the conclusion to work with the mini-cube length of 16 pixels and how I developed the algorithm for the detection of outliers (centroids) in the Decision Graph of the DP process.

Should you wish to experiment with the code developed so far, I recommend starting with the content in the Jupyter notebook files. It is more useful for further analysis than the GUI application app.py.

References

[1] Elsevier, Information Technology and Quantitative Management, *Image Segmentation via Improving Clustering Algorithms with Density and Distance*, 21 July 2015, by Zhensong Chen, Zhiquan Qi, Fan Meng, Limeng Cui, Yong Shi, https://www.sciencedirect.com/science/article/pii/S1877050915015719

[2] Python 3, programming language, latest download available at https://www.python.org/downloads

[3] Jupyter Notebook, IDE for the development of programs in Python, download available at http://jupyter.org/install.html

[4] tkinter, Graphical User Interfaces for Python programs, documentation available at https://docs.python.org/3/library/tk.html

[5] PIL, Python Image Library, useful for the processing of images; documentation available at https://pillow.readthedocs.io/en/5.1.x/reference/Image.html

[6] R Language, The R Project for Statistical Computing, documentation and downloads available at https://www.r-project.org

[7] International Journal of Advanced Computer Science and Applications (IJACS) Vol. 8, No. 9, 2017, *Face Extraction from Image based on K-Means Clustering Algorithms*, by Yousef Farhang, https://thesai.org/Downloads/Volume8No9/Paper_14-Face_Extraction_from_Image_based_on_K_Means.pdf

[8] IEEE, ISBN 1-4244-0069-4, 5 June 2006, *Medical Image Segmentation Using K-Means Clustering and Improved Watershed Algorithm,* by H.P. Ng *et al*. https://ieeexplore.ieee.org/abstract/document/1633722

[9] Elsevier, Computerized Medical Imaging and Graphics, *Fuzzy c-means (FCM) clustering with spatial information for image segmentation*, 19 December 2005, by Keh-Shih Chuang *et al*., https://www.sciencedirect.com/science/article/pii/S0895611105000923

[10] Nagoya University, *Unsupervised Pathology Image Segmentation using representation learning with spherical K-Means*, 11 April 2018, by Takayasu Moriya *et al*., https://arxiv.org/abs/1804.03828

[11] 11th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition (EMMCVPR), *A Convex Approach to K-means Clustering and Image Segmentation*, Oct 2017, Venice, Italy, by Laurent Condat, Oct 2017, Venice, Italy. 2017, https://hal.archives-ouvertes.fr/hal-01504799/document

[12] International Journal of Computer Trends and Technology (IJCTT), Volume 4, Issue 5, *Image segmentation using K-mean clustering for finding tumor in medical*

application, May 2013, by Piyush M. Patel *et al.*,
http://ijcttjournal.org/archives/ijctt-v4i5p55

[13] Anaconda Distribution of Python 3 and Jupyter Notebook, *Download page of Anaconda for Windows, Mac OSX and Linux*,
https://www.anaconda.com/download

[14] Python Software Foundation, *Built-in Types - Dictionaries*,
https://docs.python.org/3/library/stdtypes.html#dict

[15] NumPy, *Fundamental package for scientific computing with Python*,
http://www.numpy.org

[16] Scikit-learn, *Machine Learning in Python*, http://scikit-learn.org/stable