HDDA Challenge Final Report

Walter Fenske, Markos Meytarjyan, Anh Phan

2/29/2024

Introduction

In this project we will be using vector quantization along with a dataset of cat images to show how image compression works. We will define all the required functions, including transfering a matrix into blocks and clustering them using k-means clustering, converting the compressed image into right format, and then try to analyze optimal parameter for compression using distance from original image.

1. What is Vector Quantization and How Can We Apply It?

Quantization in this context is the subdivision of something (mainly a matrix) into something smaller that has measurable attributes. In essence, we will take a matrix that represents an image of a cat and subdivide it into smaller blocks. We will then perform k-means clustering on all the blocks that we have divided (2×2 blocks for our project specifically). Finally, we will take that matrix and upscale it with varying k in k-means. In the context of image processing, this technique can be applied to reduce the number of distinct pixel values, thereby compressing the image (reduce the amount of storage required for an image).

2. Application of quantization on compressing cat images

In this project, we will apply the quantization method with different number of clusters k into compressing cat images, and try to visualize the image from the compressed version. First of all, we will need to import the dataset and conduct some preliminary transformation to convert our dataset of cat images into right form.

```r
cats_tbl <- read_csv("~/Stats 382 S24/Class/Data/cats2.csv")
image_matrix <- cats_tbl %>%
  select (-c(rowid)) %>%
  as.matrix()
rownames(image_matrix) <- cats_tbl$rowid
```

In the dataset table, we have 99 rows representing 99 cats, as well as 4096 columns, where each column is a pixel value for the image of that cat (all the cat images in our dataset would be in the form of 64×64). Now we will look at how we can quantize a cat image with a certain number of cluster k.

a. Make quantized image using k-means cluster (K-means Image Function)

Now we will make a function that inputs a row of pixel values from the dataset "image_matrix", and returns "kmeans" object as the quantized image (since we need the centroid of all clusters, as well as what is the cluster that each blocks belong to). We first reshape a numeric vector into a matrix with n rows, filling the matrix row by row to respect the original data structure. Then, we would need to extracts 1024 2x2 blocks from the image and creates a new matrix with 1024 rows and 4 columns. Why 4 columns? 2x2 = 4, we need an entry for each value of the pixel (order of the pixels when we input into the dataframe would be: top-left, top-right, down-left, down-right, and we iterate through the top-left corner of each block). Finally, it creates a k means object with the new 1024 x 4 matrix, with the parameters k to be the number of clusters that we inputed, and our function will return a kmeans object as the compressed image. Here are the "kmeans_image" function that follows our scheme as above:

```r
kmeans_image <- function(row_image, n, k) {
  # Here, a row of pixel will first turns into an image
  image <- matrix(as.numeric(row_image),nrow=n,byrow=T)
  # Now we will extract all 2*2 blocks
  # and make this into a matrix of 4 columns and 1024 rows
  image_blocks <- matrix(0, nrow = length(row_image) / 4, ncol = 4)
  for (y in 1:(n/2)) {
    for (x in 1:(n/2)) {
      # insert blocks with upper-left corner: (2x-1, 2y-1) into row (n/2)*(y-1) + x
      image_blocks[x + (n/2)*(y - 1), 1] <- image[2*x-1, 2*y-1]
      image_blocks[x + (n/2)*(y - 1), 2] <- image[2*x-1, 2*y]
      image_blocks[x + (n/2)*(y - 1), 3] <- image[2*x, 2*y-1]
      image_blocks[x + (n/2)*(y - 1), 4] <- image[2*x, 2*y]
    }
  }
  # Now we perform kmeans clustering
  k_clusters <- kmeans(image_blocks, centers = k, iter.max = 1000)
  return(k_clusters)
}
```

We can quickly test this with image of cat 10 to see if this function runs properly:

```r
set.seed(12345)
test_cluster <- kmeans_image(image_matrix[10, ], 64, 10)
class(test_cluster)
```

```
## [1] "kmeans"
```

We can test out this function by taking a look at the number of elements at each cluster and the heatmap of cluster centers using different values of k.

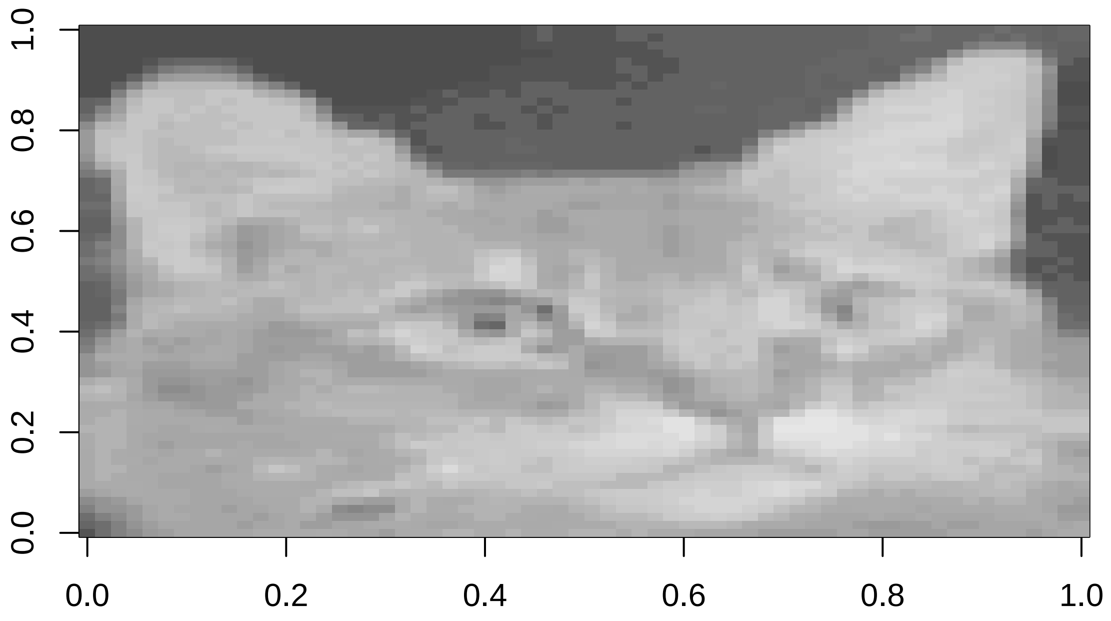
b. Inspecting the clustering by kmeans_image function.

First we will need a function to visualize the original image, by converting the row of pixel into a square matrix of pixel and then visualize it using image() in R:

```r
plotImage <- function(dat,size=64){
  imag <- matrix(as.numeric(dat),nrow=size,byrow=T)
  image(imag,col=grey.colors(256))
}
```

Below we can see that this function works since we can print the image of the first cat:

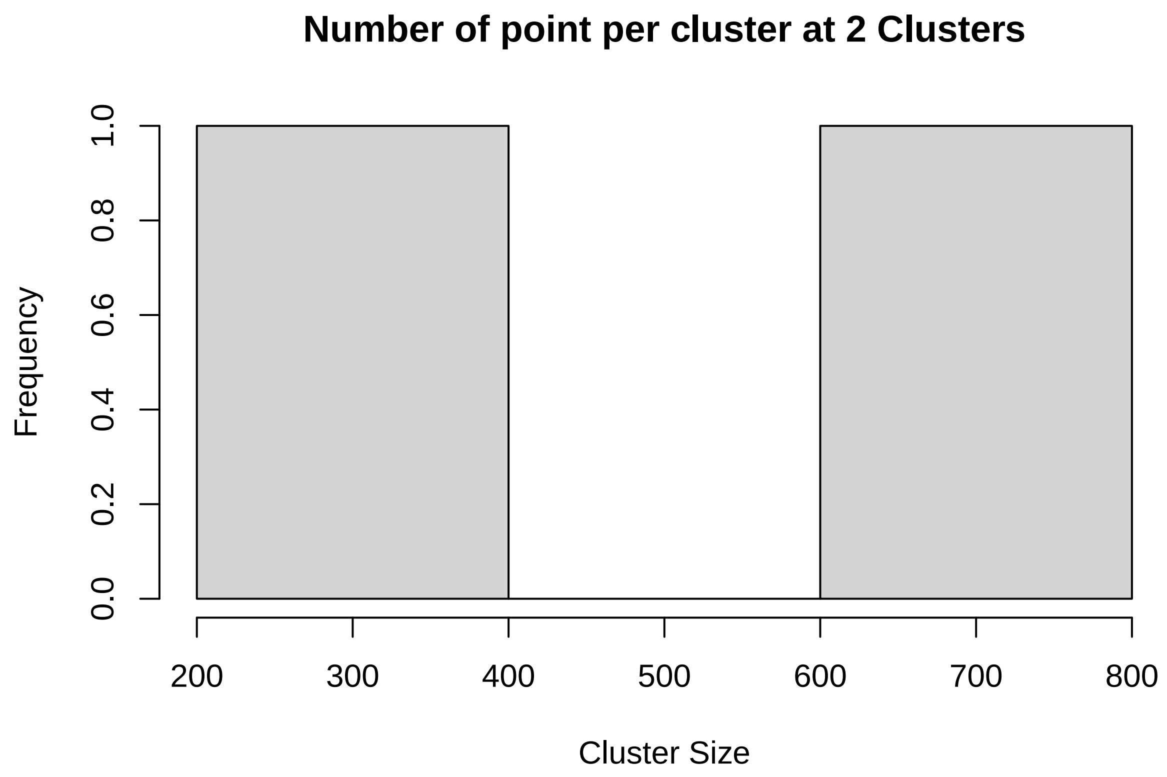
```
plotImage(image_matrix[1,])
```

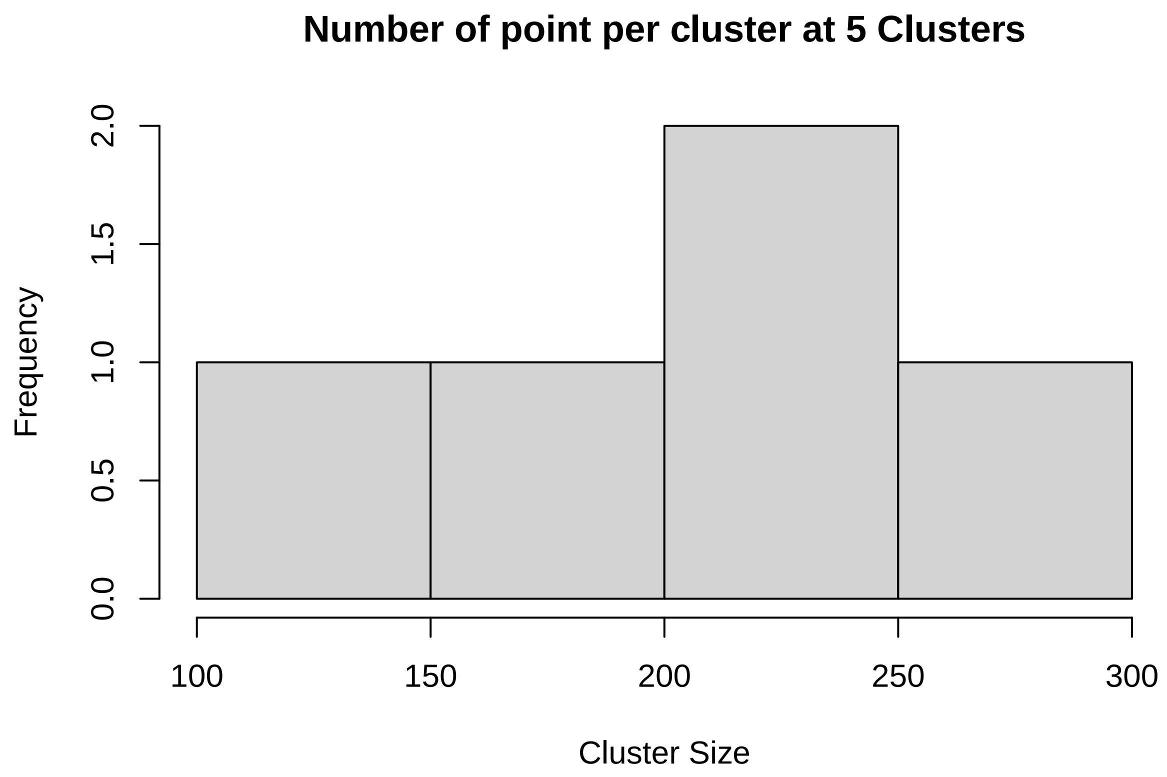


Plot the histogram of number of observations for $k = 2, 5, 10, 50, 100$ clusters

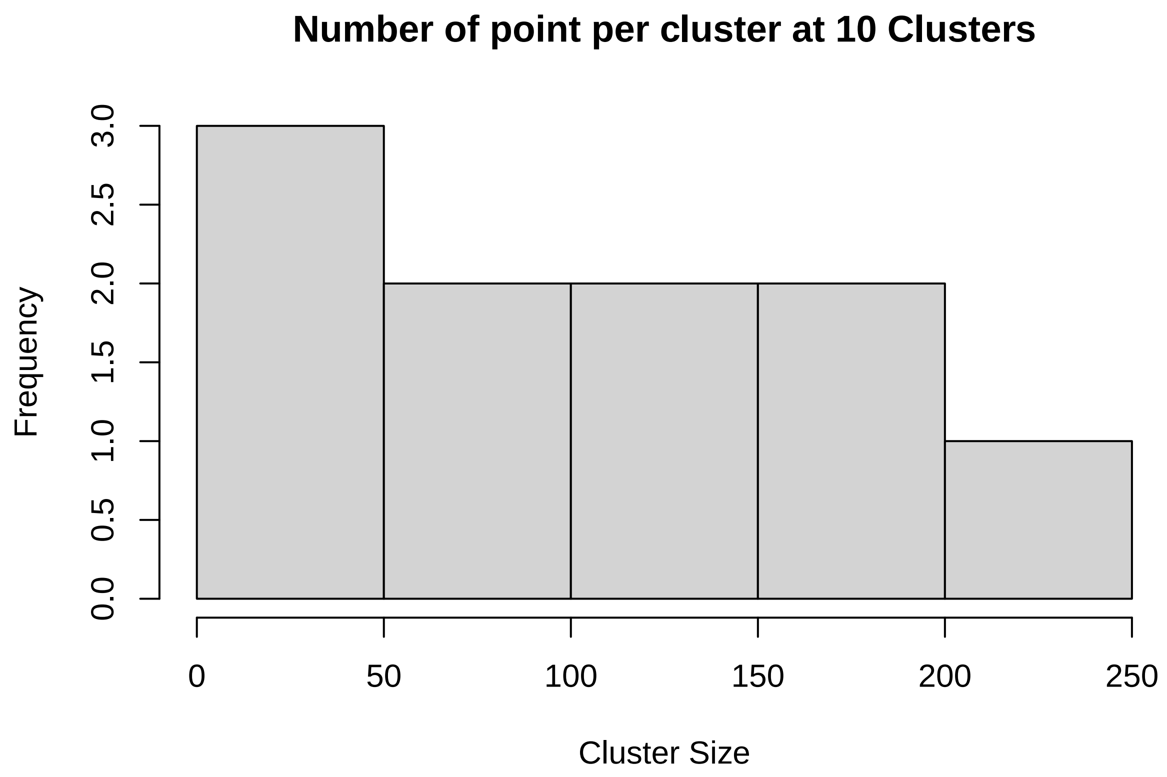
We first look at cat 1 and the histogram with each value of k similar to the values that we considered above:

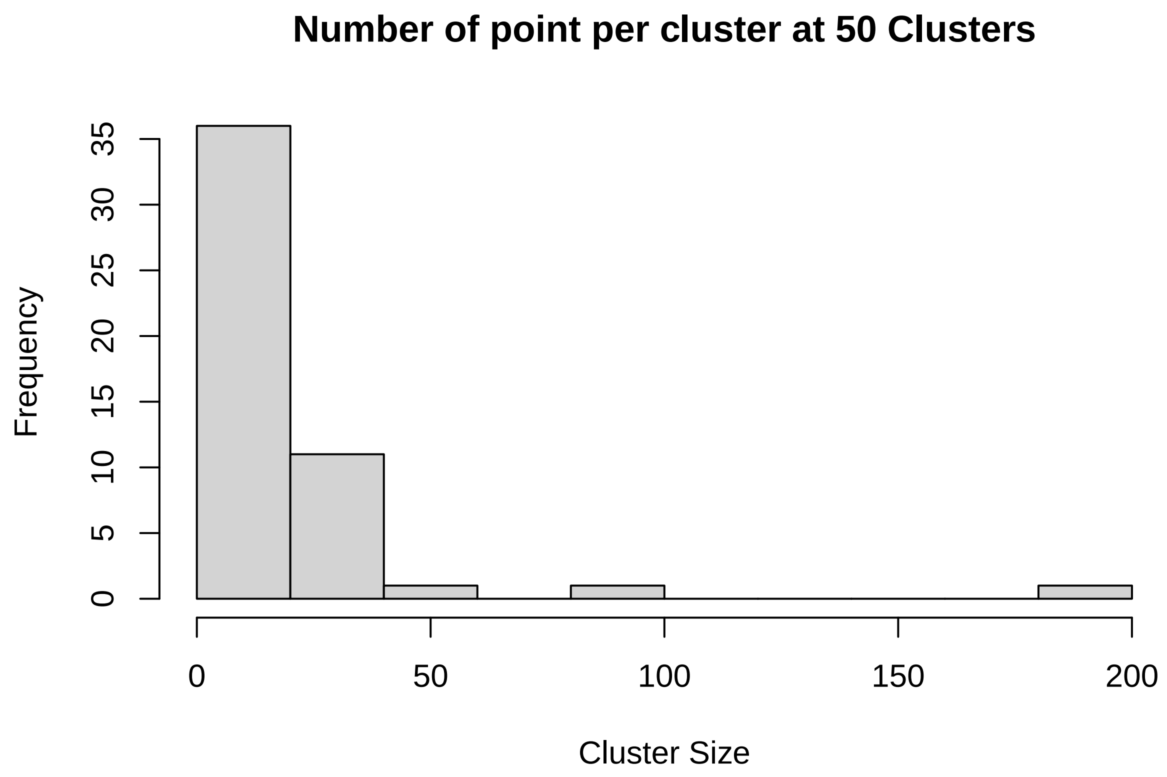
```
set.seed(12345)
# test for 2, 5, 10, 50, 100 cluster and draw the correspond histogram
for (i in c(2, 5, 10, 50, 100)) {
  image_cluster <- kmeans_image(image_matrix[1, ], 64, i)
  hist(image_cluster$size,
        main = paste("Number of point per cluster at", as.character(i), "Clusters"),
        xlab = "Cluster Size", ylab = "Frequency")
}
```

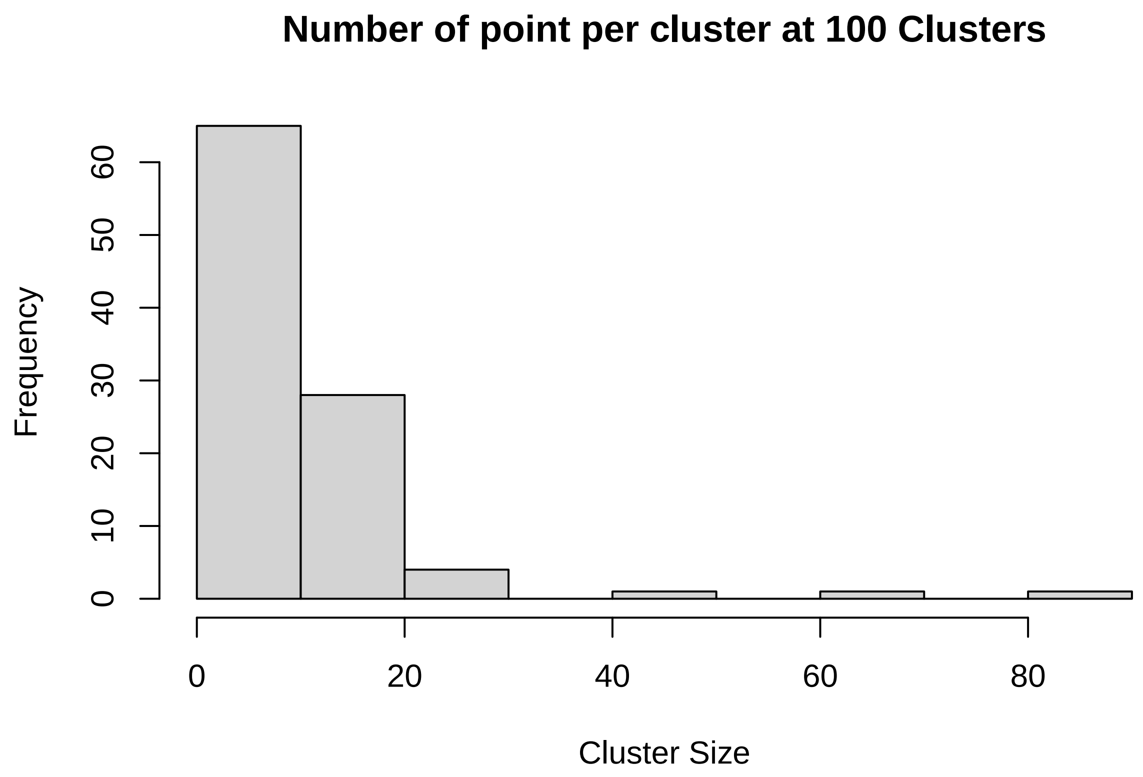

Number of point per cluster at 2 Clusters
Frequency
0.0
0.2
0.4
0.6
0.8
1.0
200
300
400
500
600
700
800
Cluster Size



Number of point per cluster at 5 Clusters
Frequency
0.0
0.5
1.0
1.5
2.0
100
150
200
250
300
Cluster Size


Number of point per cluster at 10 Clusters
Frequency
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
50
100
150
200
250
Cluster Size


Number of point per cluster at 50 Clusters
Frequency
0
5
10
15
20
25
30
35
0
50
100
150
200
Cluster Size



For small value of k such as 2, 5, we can see that the number of points in each cluster can differ quickly (especially for 5 when we have cluster size to vary from under 100, under 200, to above 300), which can be explained that the image only have a few colors, with some light color to be the major color of the image, and since we divide the image into small blocks, there is high chance that each block only have 1 color, which leads to this. As we goes to higher k value, the clustering process would be able to represent different part of the image much easily, and so the number of points in each cluster are smaller since the blocks that might be from same clusters for small k might now (for 100 clusters, about 65 clusters have less than 5 blocks inside it, or for 50 clusters, more than half of the clusters have less than 15 blocks)

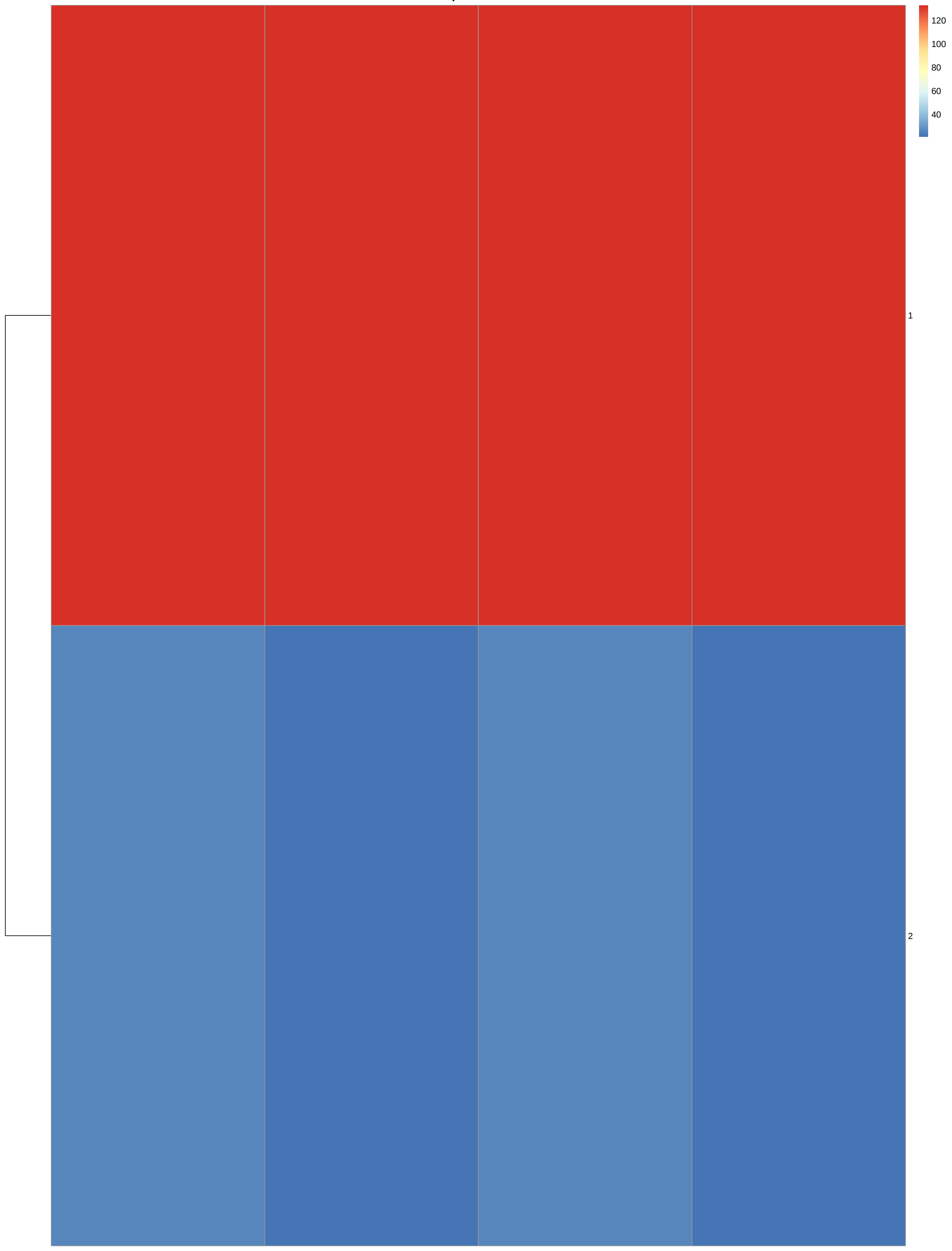
From here, we can also look at the heatmap of the centroid for different number of clusters

Plot the centroid heatmap when we have $k = 2, 5, 10, 50, 100$ clusters

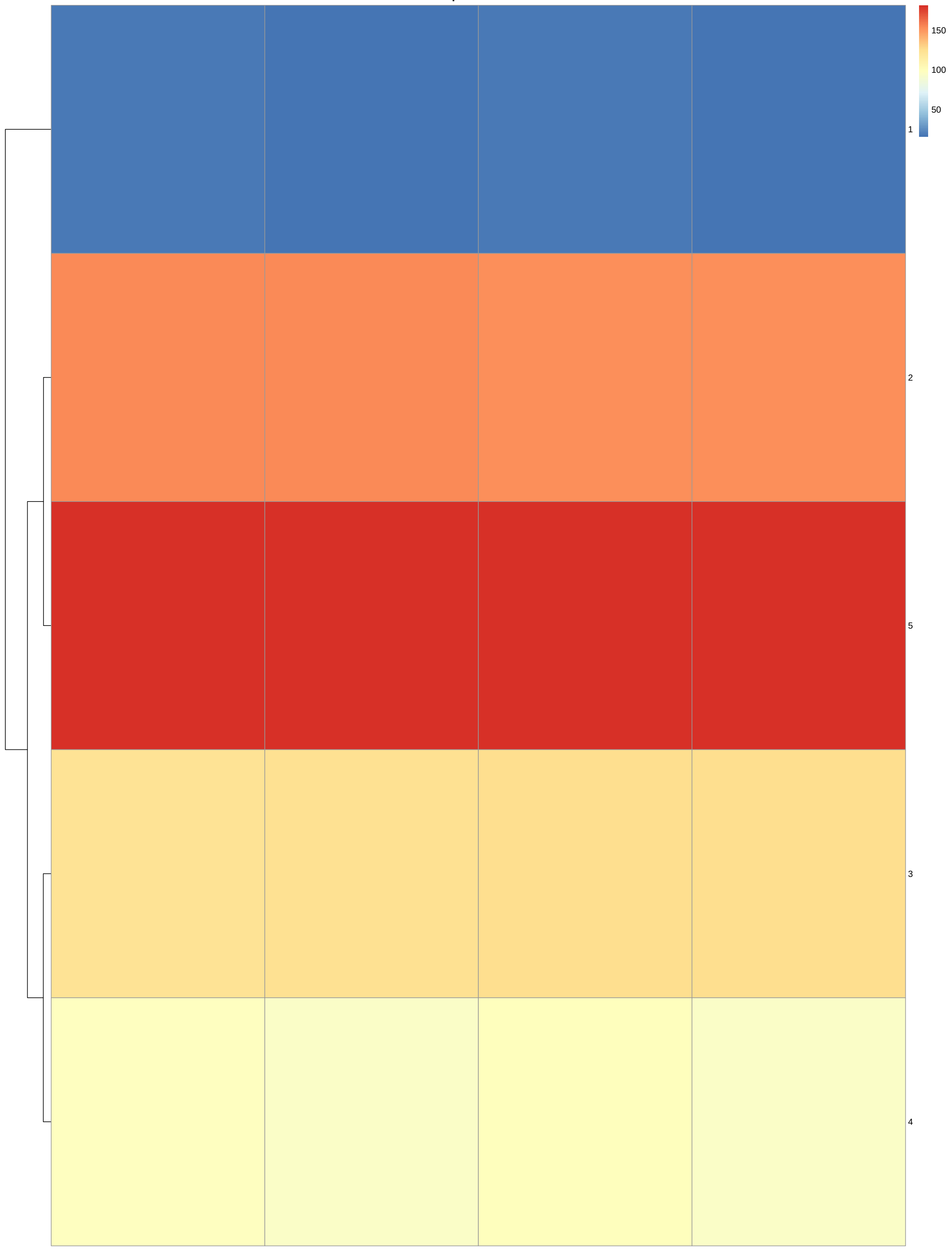
We first look at the heatmap of the centroids for different numbers of k for cat 1:

```
set.seed(12345)
# test for 2, 5, 10, 50, 100 cluster and draw the correspond heatmap
for (i in c(2, 5, 10, 50, 100)) {
  image_cluster <- kmeans_image(image_matrix[1, ], 64, i)
  pheatmap(image_cluster$centers, cluster_cols = FALSE, cluster_rows = TRUE,
           main = paste("For cat 1: Heatmap of the center of", as.character(i), "clusters"))
}
```

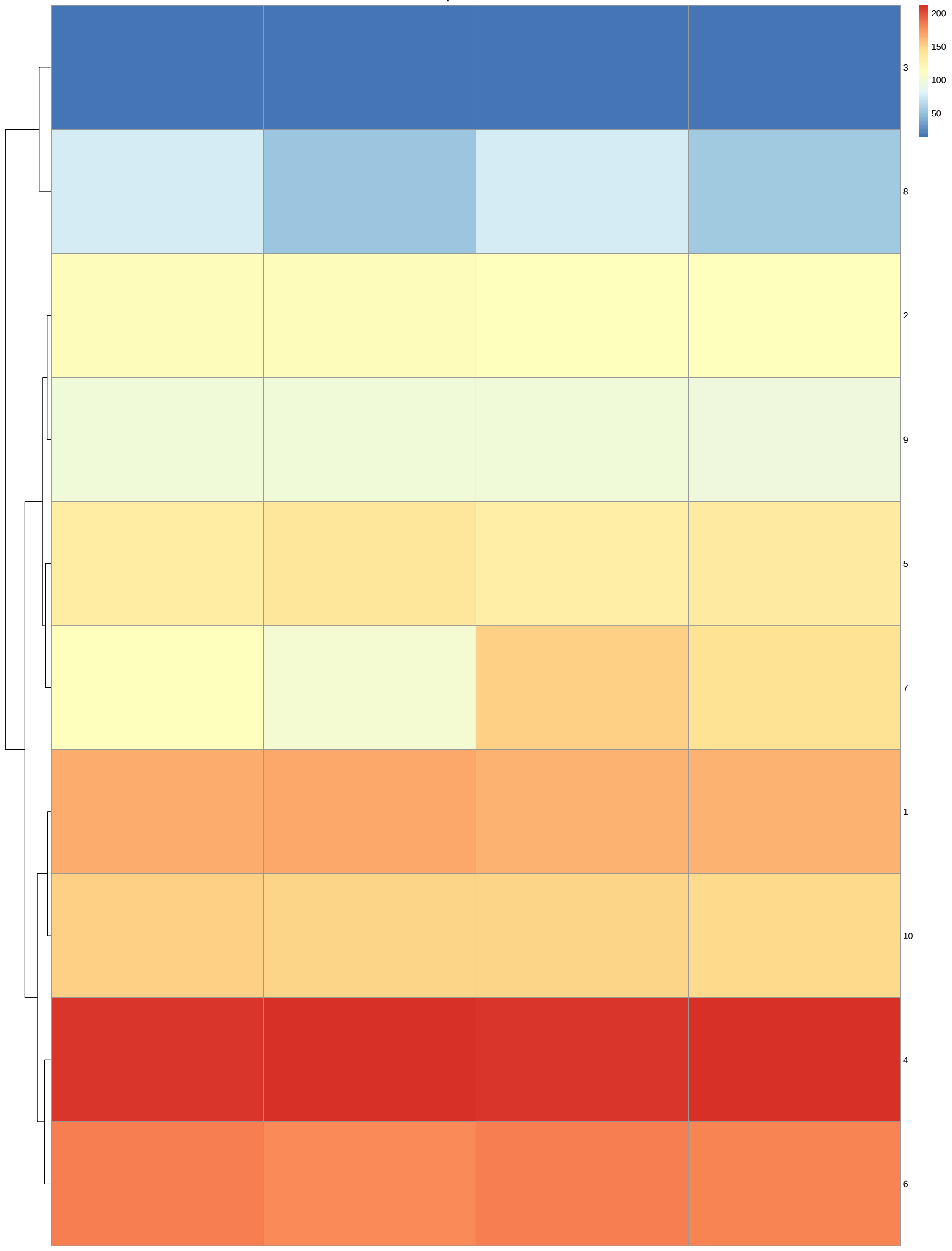
For cat 1: Heatmap of the center of 2 clusters



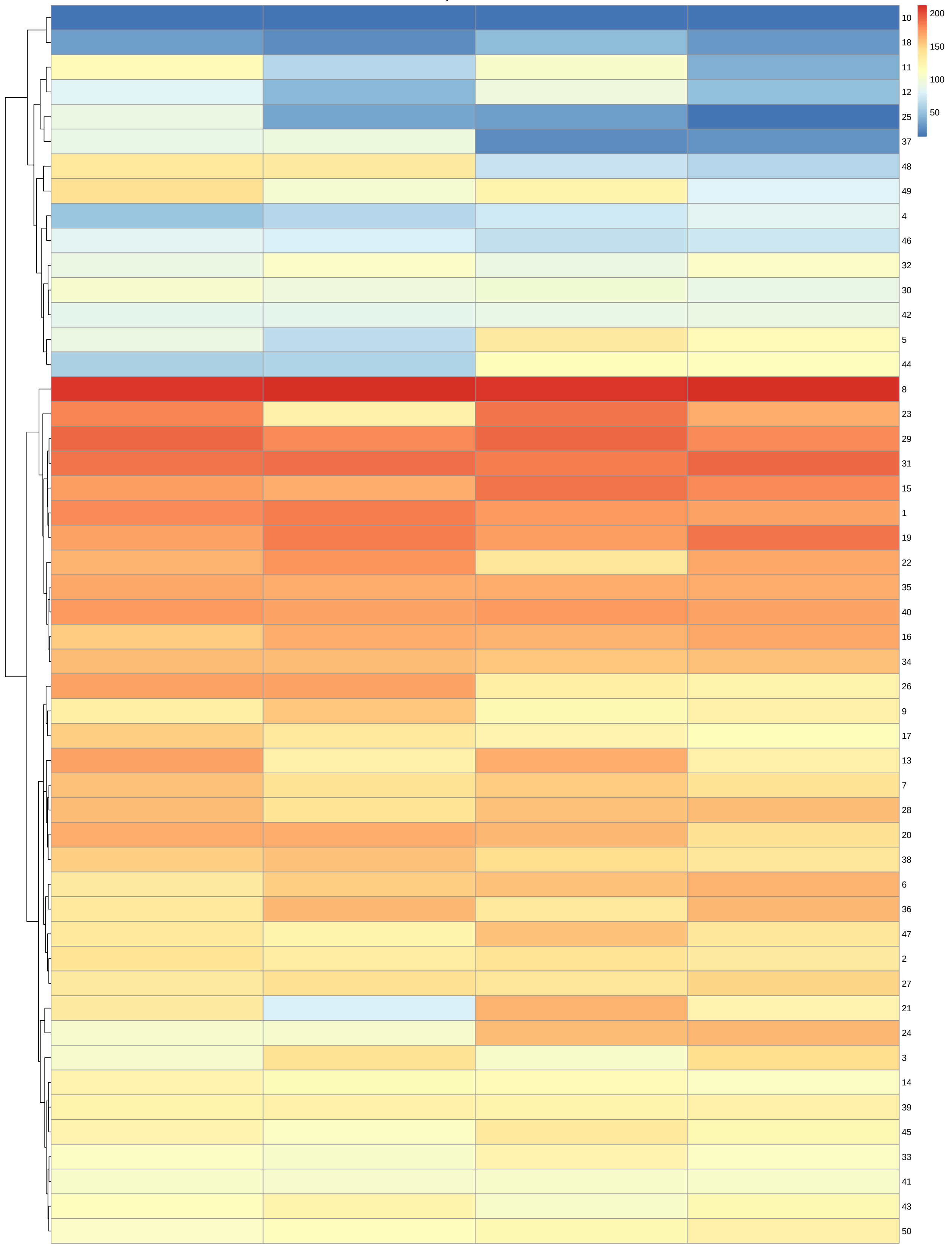
For cat 1: Heatmap of the center of 5 clusters

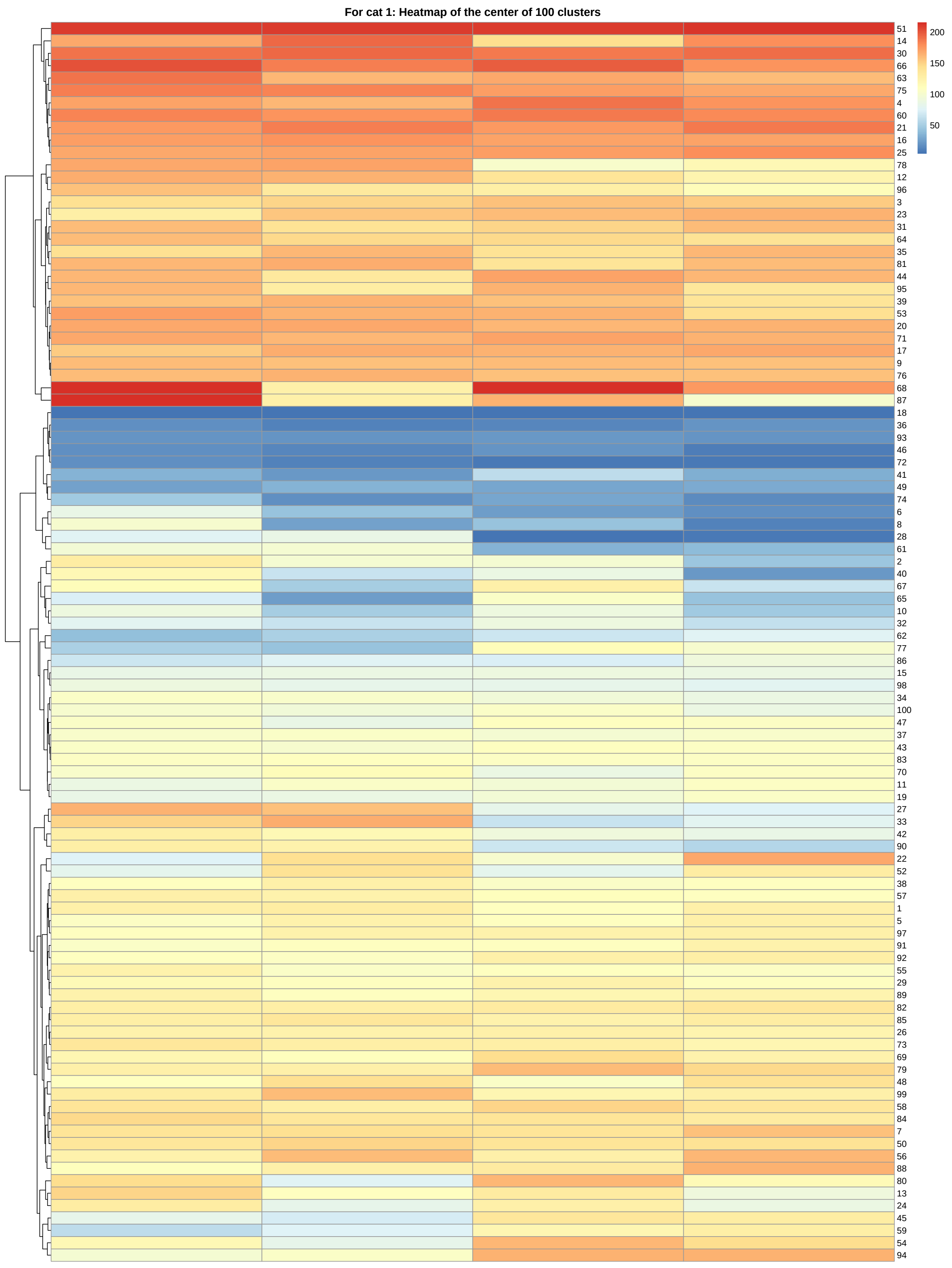


For cat 1: Heatmap of the center of 10 clusters



For cat 1: Heatmap of the center of 50 clusters





As we can see above for up to 10 clusters it is quite easy to see that the clusters withing themselves are well separated, we could also observe that for clusterings with $k = 50, 100$ however there are two many rows and it is not as simple to have a good visual outlook. Another thing we can notice is that for $k = 100$ different clusters have values from almost entire spectrum of the legend which is generally good meaning that different colors could be represented in the compressed image. It is generally not that easy to evaluate the performance of the compression without looking at images directly, which we will take care of in the next section.

3. Image compression in practice

After this, we can see that we have successfully made our function, but one problem arises, which is we do not have any function in R that can visualize our image so that we can see how the quality changes comparing to the original image. This brings us to our second step: making a function that allow us to visualize a compressed image as a kmeans object.

a. Making the image compression function (compress_image)

Similar to kmeans_image them to make the compress_image function that compress an image using quantization and return a matrix of pixel for the quantized image. We can use the kmeans_image function inside our compress_image function to get the kmeans clustering stored in a variable. Afterwards we create an empty matrix where we will store the pixel values of the compressed image. We iterate through all the clusters of kmeans clustering and fill each block with the pixel of the centroid found during clustering (in the same manner as when we put the pixels in each 2×2 blocks into a dataframe for clustering). We can implement it in R as follow:

```
compress_image <- function(row_image, n, k) {
  # We perform kmeans clustering using kmeans_image
  image_cluster <- kmeans_image(row_image, n, k)

  imag <- matrix(data = 0, nrow = n, ncol = n)
  for (i in 1:length(image_cluster$cluster)) {
    # find the location of top left corner (2x - 1, 2y - 1)
    if (i %% (n / 2) == 0) {
      x <- n / 2
    } else {
      x <- i %% (n / 2)
    }
    y <- ((i-x) %/% (n / 2)) + 1
    # fill each block with the pixel of the centroid
    curr_cluster <- image_cluster$cluster[[i]] # find the cluster id
    curr_centroid <- image_cluster$centers[curr_cluster, ]
    imag[2*x-1, 2*y - 1] <- curr_centroid[[1]]
    imag[2*x-1, 2*y] <- curr_centroid[[2]]
    imag[2*x, 2*y-1] <- curr_centroid[[3]]
    imag[2*x, 2*y] <- curr_centroid[[4]]
  }

  return(imag)
}
```

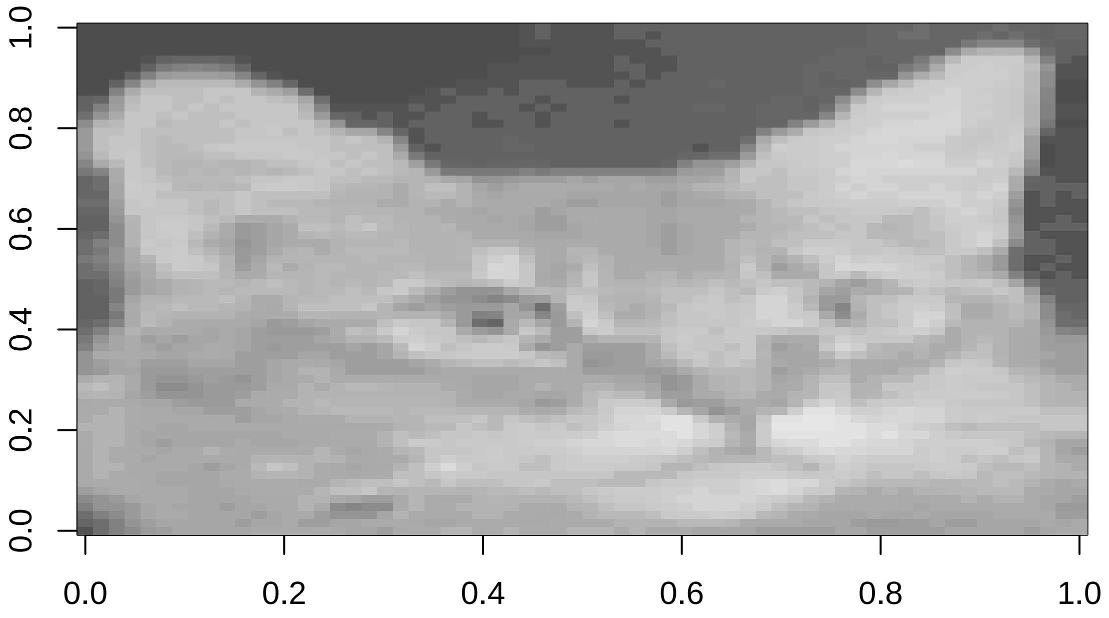
After this, we can test to see if our function returns the right type of output:

```r
set.seed(12345)
test_quantized_image <- compress_image(image_matrix[10, ], 64, 10)
class(test_quantized_image)
```

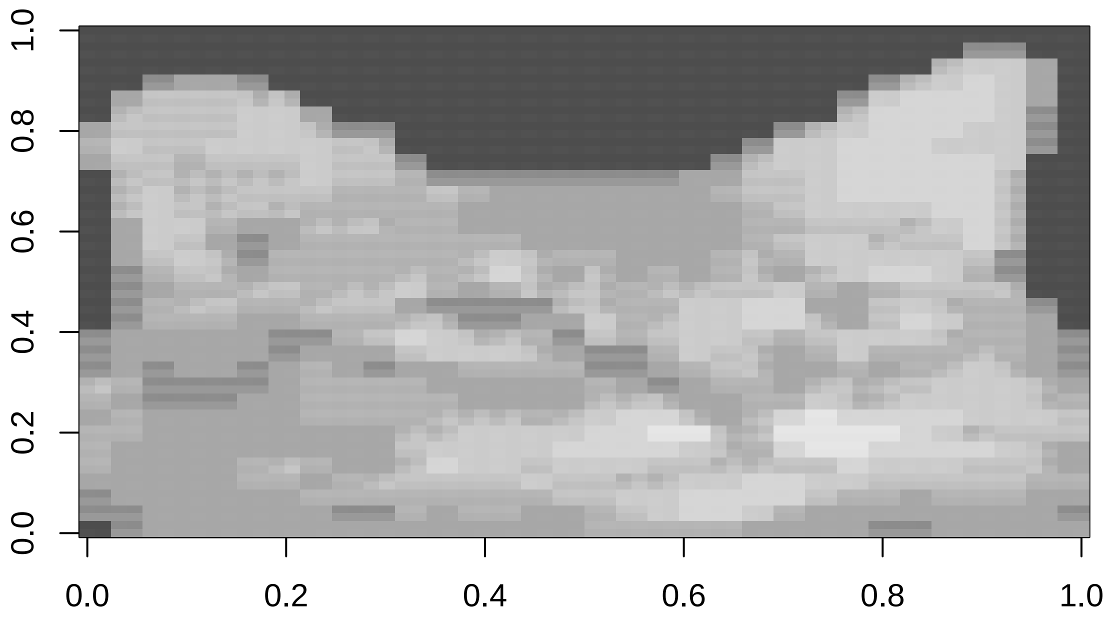
```
## [1] "matrix" "array"
```

We can also try to print out a quantized image of the first cat to make sure the functions works:

```r
cat1_row <- c(image_matrix[1,])
plotImage(cat1_row)
```



```r
plotImage(t(compress_image(cat1_row, 64, 10)))
```



b. Testing the compress_image function

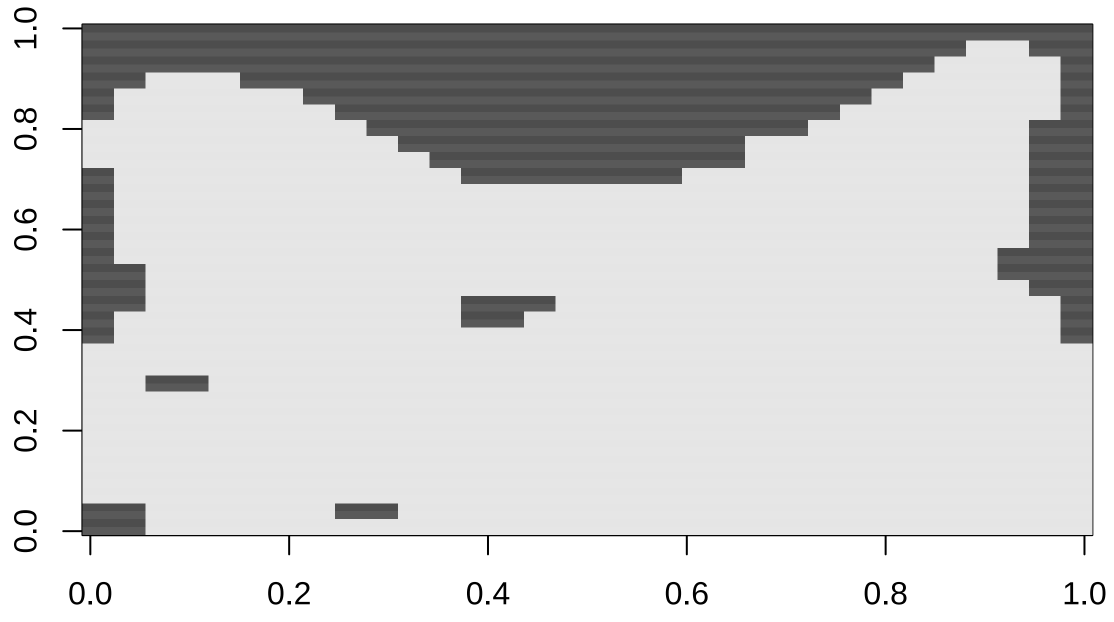
We can now take a proper look at the image compression and see how it works based on images of two different cats and image compression for different values of k. In this part, we will use two cats that have different fur color in order to compare the effect our the compression on different type of images. The two cats that we will use are cat 1 and 50. We can visualize these two cat to see how they originally look like.

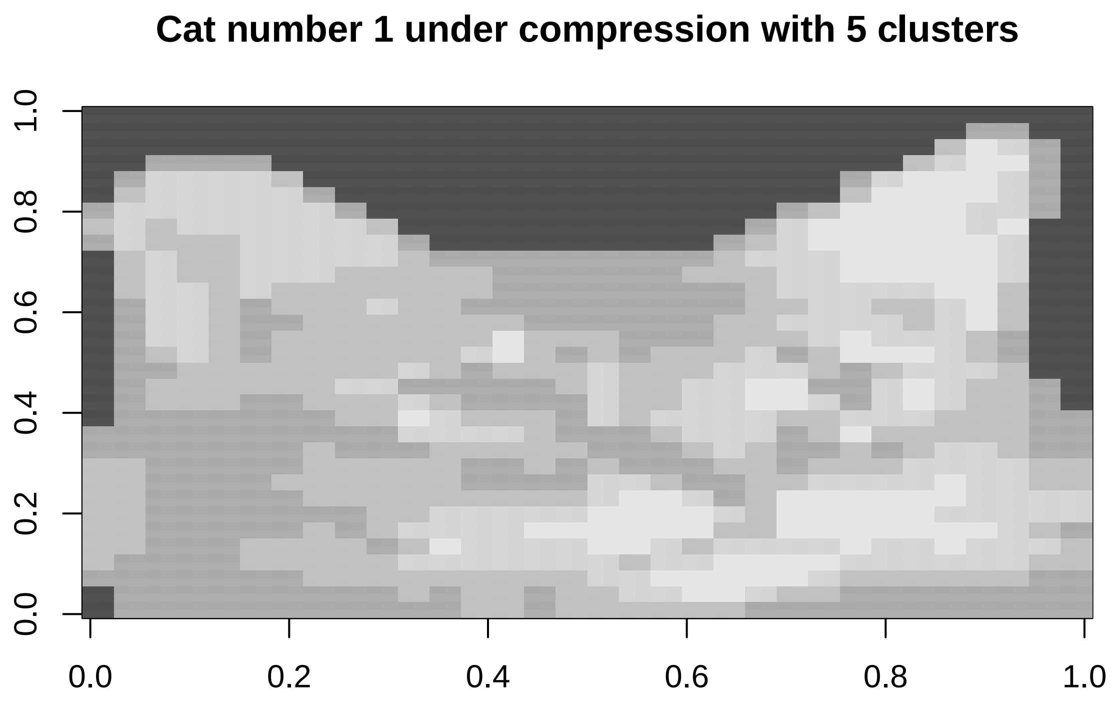
Plot the images of cats number 1 and 50 for $k = 2, 5, 10, 50, 100$ clusters

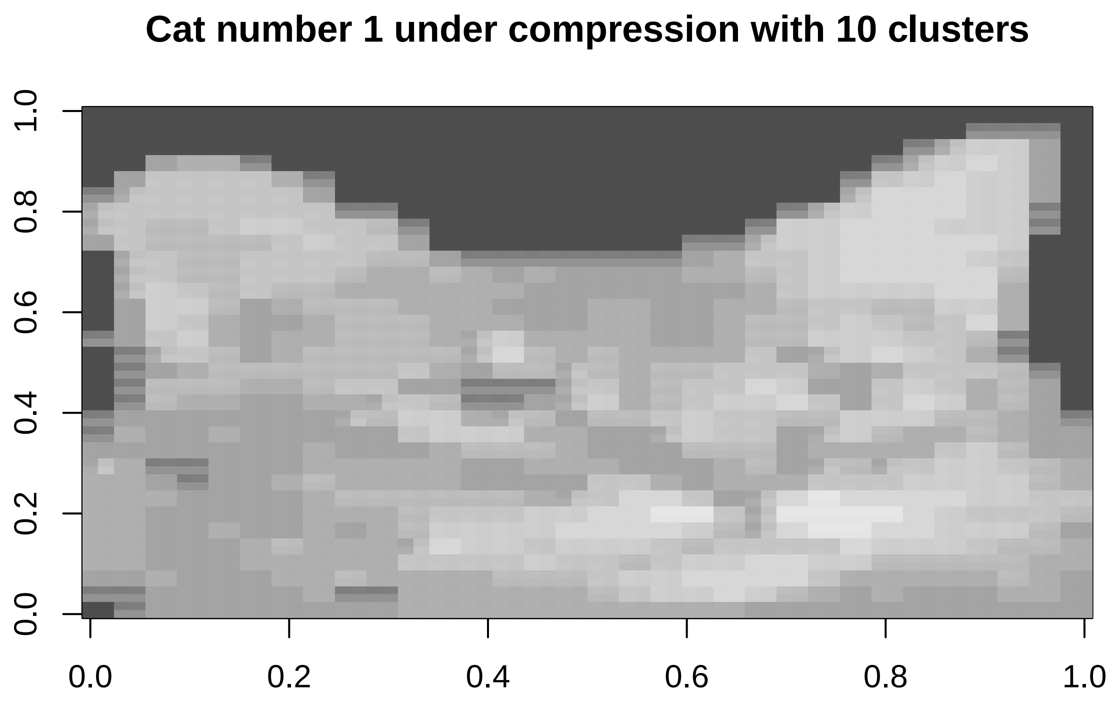
We first look at cat 1 and how did it changes:

```r
set.seed(12345)
# test for 2, 5, 10, 50, 100 clusters and draw the image
for (i in c(2, 5, 10, 50, 100)) {
  plotImage(t(compress_image(image_matrix[1,], 64, i)))
  title(paste("Cat number 1 under compression with", as.character(i), "clusters"))
}
```

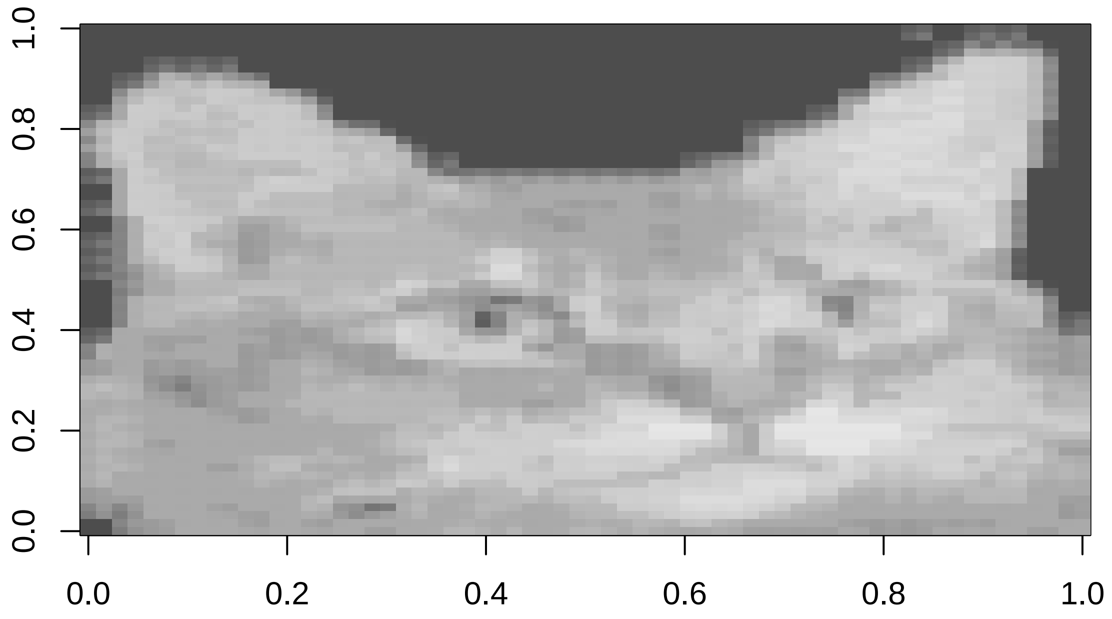
Cat number 1 under compression with 2 clusters

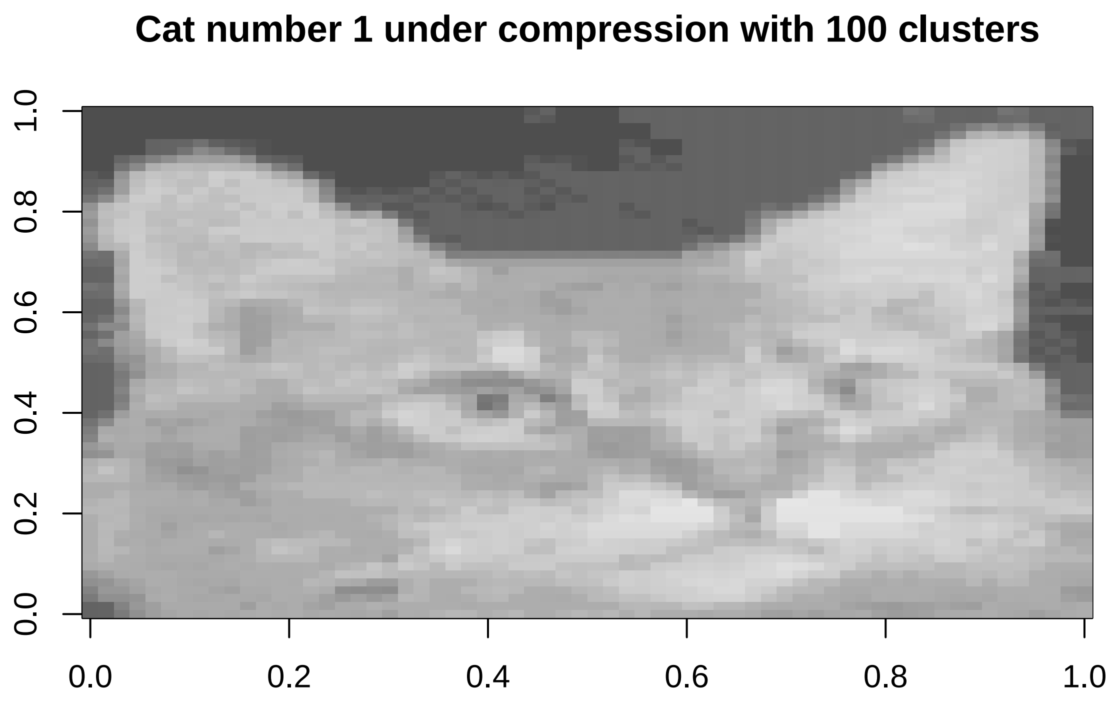



Cat number 1 under compression with 5 clusters
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0


Cat number 1 under compression with 10 clusters
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0

Cat number 1 under compression with 50 clusters

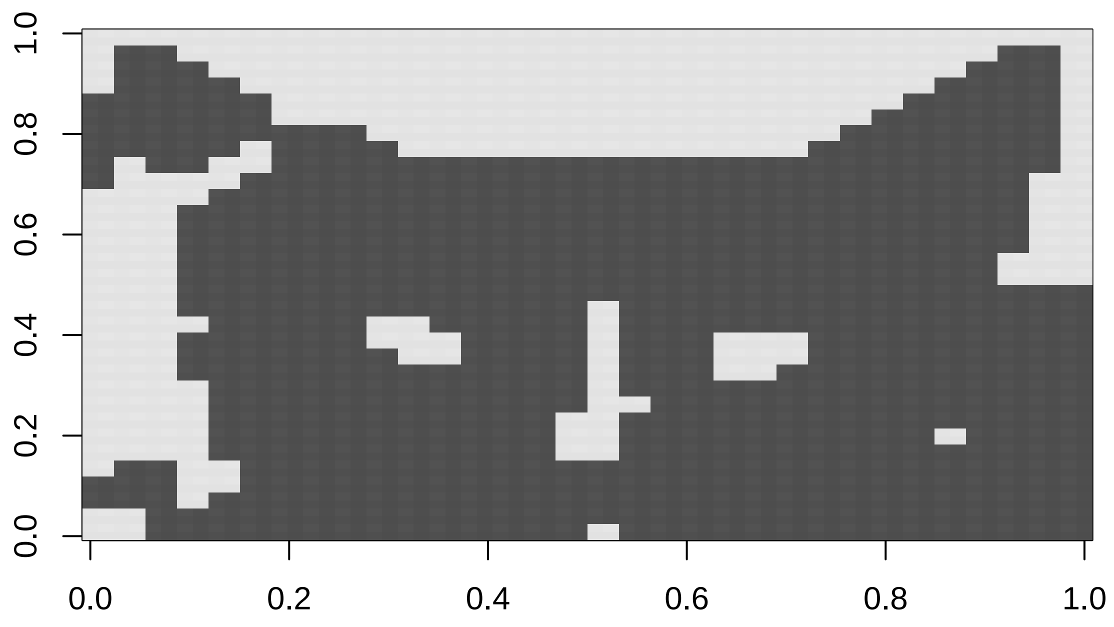


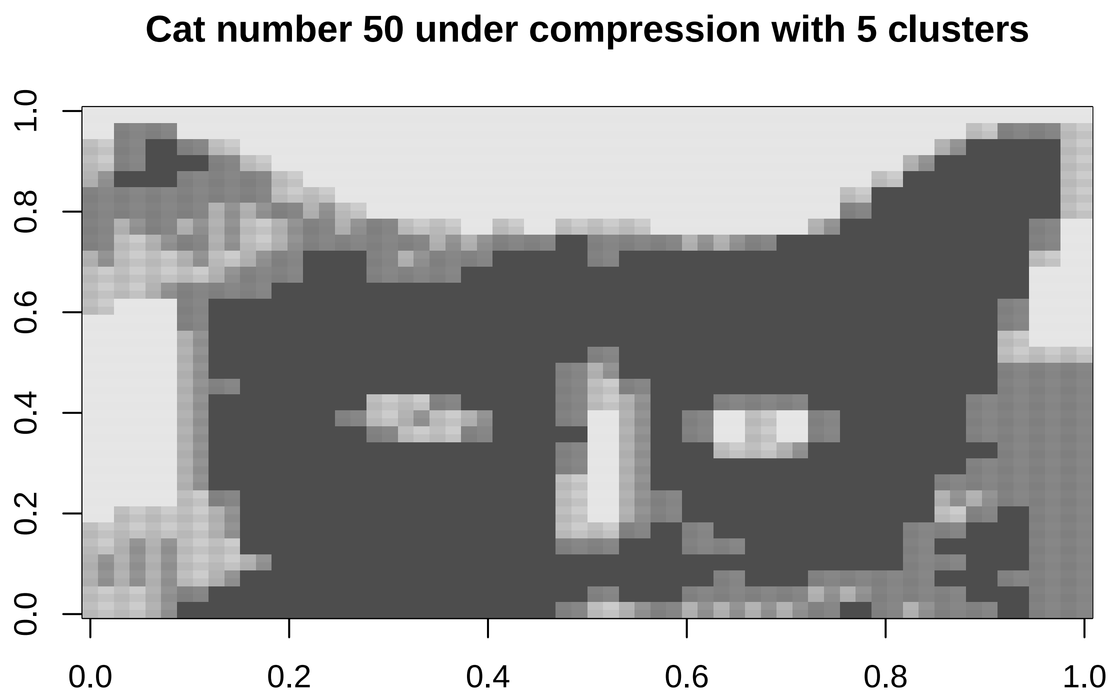


Now we can look at the image of cat 50 when compressed with different numbers of clusters:

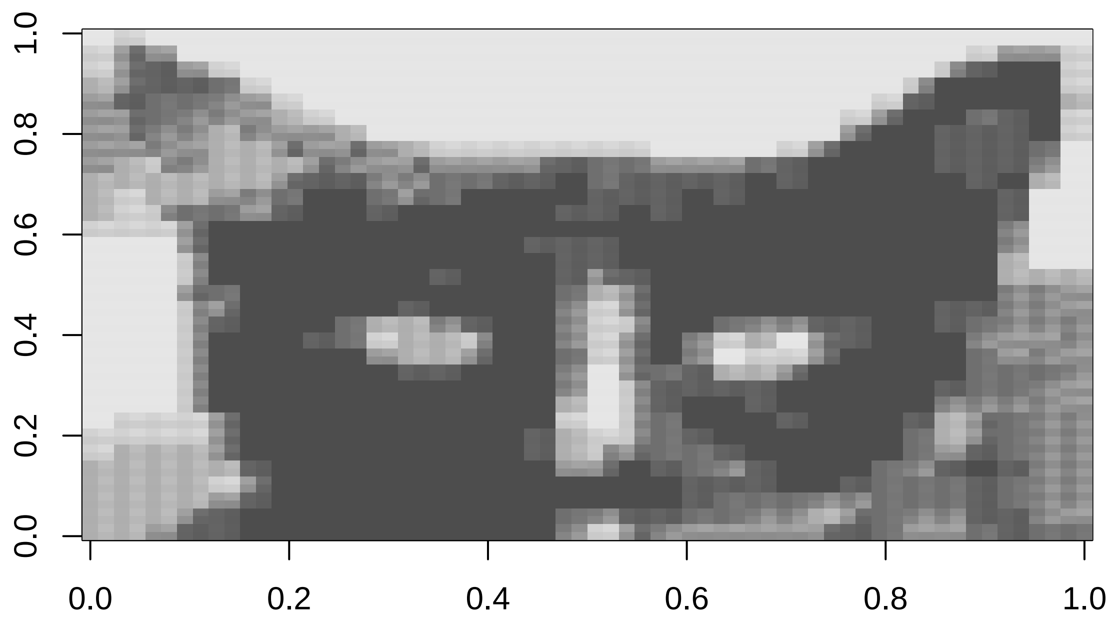
```
set.seed(12345)
# test for 2, 5, 10, 50, 100 clusters and draw the image
for (i in c(2, 5, 10, 50, 100)) {
  plotImage(t(compress_image(image_matrix[50,], 64, i)))
  title(paste("Cat number 50 under compression with", as.character(i), "clusters"))
}
```

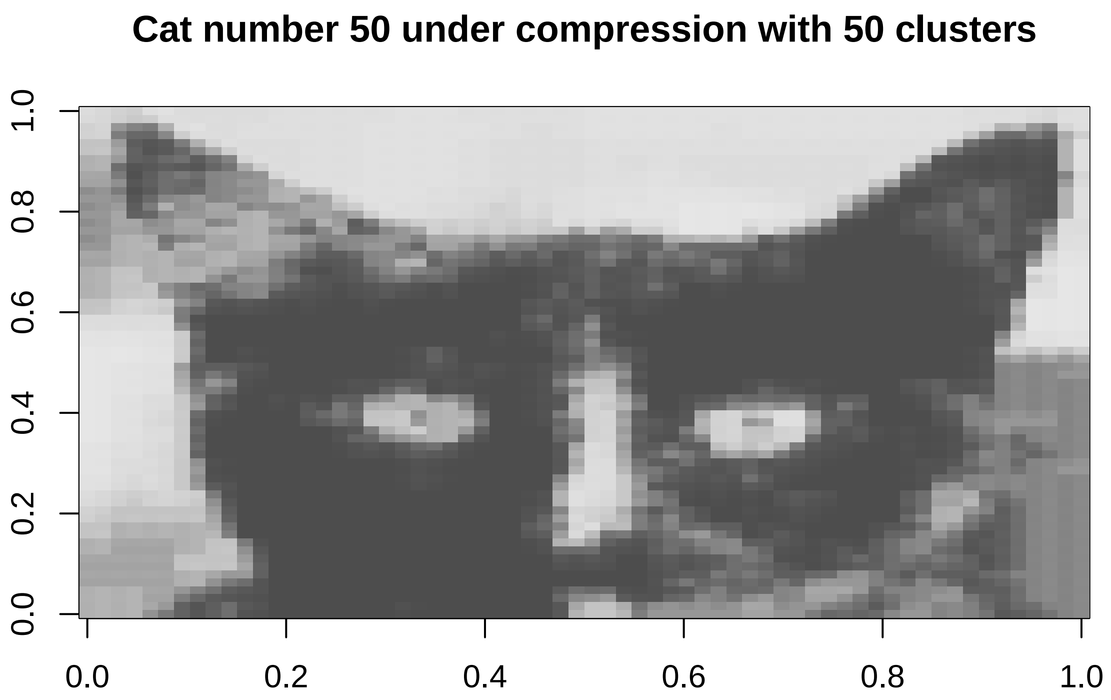
Cat number 50 under compression with 2 clusters

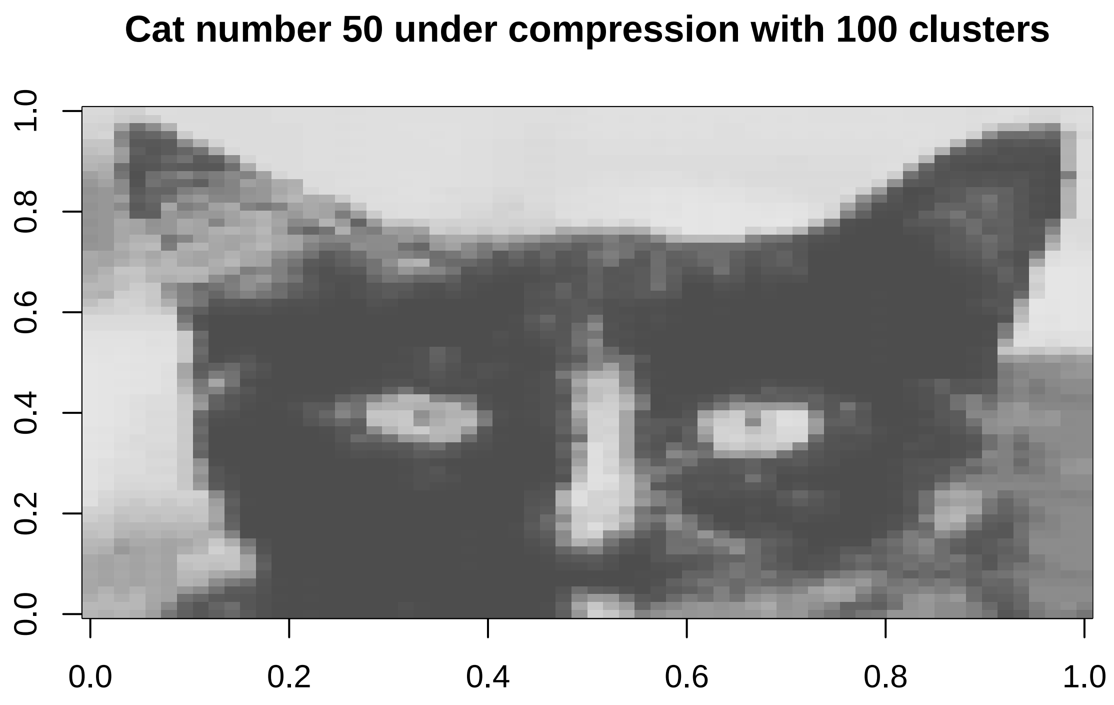



Cat number 50 under compression with 5 clusters
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

Cat number 50 under compression with 10 clusters




Cat number 50 under compression with 50 clusters
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0



As we can see above with value of $k = 2$ it is quite hard to tell anything however we can see that the compression effecively differenciatttes between background and the object. We can see that with the value $k = 5$ the image is starting to take more shape and we can see the outline of the face of the cat, however it is still extremely blurry. With higher values of k the image starts getting more clarity and already by the value $k = 100$ it is possible to tell that it is a cat without the context on the problem however even for that high number of clusters the image is still quite pixelated/distorted.

4. Image comparison (image_dist function)

After all these works, we can see that one problem would appear for the quantization process: What is the optimal number of clusters that we can use so that the quality of our image would not be decreased too much. First of all, we would need a metric on the quality of the compressed image comparing to the original image. One easy metric that we can look at is the difference between the two image matrix, calculated by the Euclidean difference of the two image matrix of the original and compressed image. By the formula of the Euclidean distance, we would need to calculate the squared root of the sum of the squared difference of the entries of two matrices. From here, we can create the function image_dist that takes on the square image of two matrices, calculates the difference of the two matrices, and finally calculates the square root of the sum of squared entries of that matrix of difference. We can implemented it in R as follow:

```r
image_dist <- function(image1, image2) {
  return(sqrt(sum((image1 - image2)^2)))
}
```

We can test our function with two custom matrix (must be same size):

```r
test_mat_1 <- matrix(data = c(1, 2, 3, 4), nrow = 2)
test_mat_2 <- matrix(data = c(7, 8, 9, 10), nrow = 2)
image_dist(test_mat_1, test_mat_2) # should show 12
```

```
## [1] 12
```

5. Choosing optimal value of k

Now we have made the function that compares two given images which means that we can asses how much the image changes depending on the value of k. We can try compressing the image for several different values of k and see when the difference between two images becomes small enough to just stop copressing at that point.

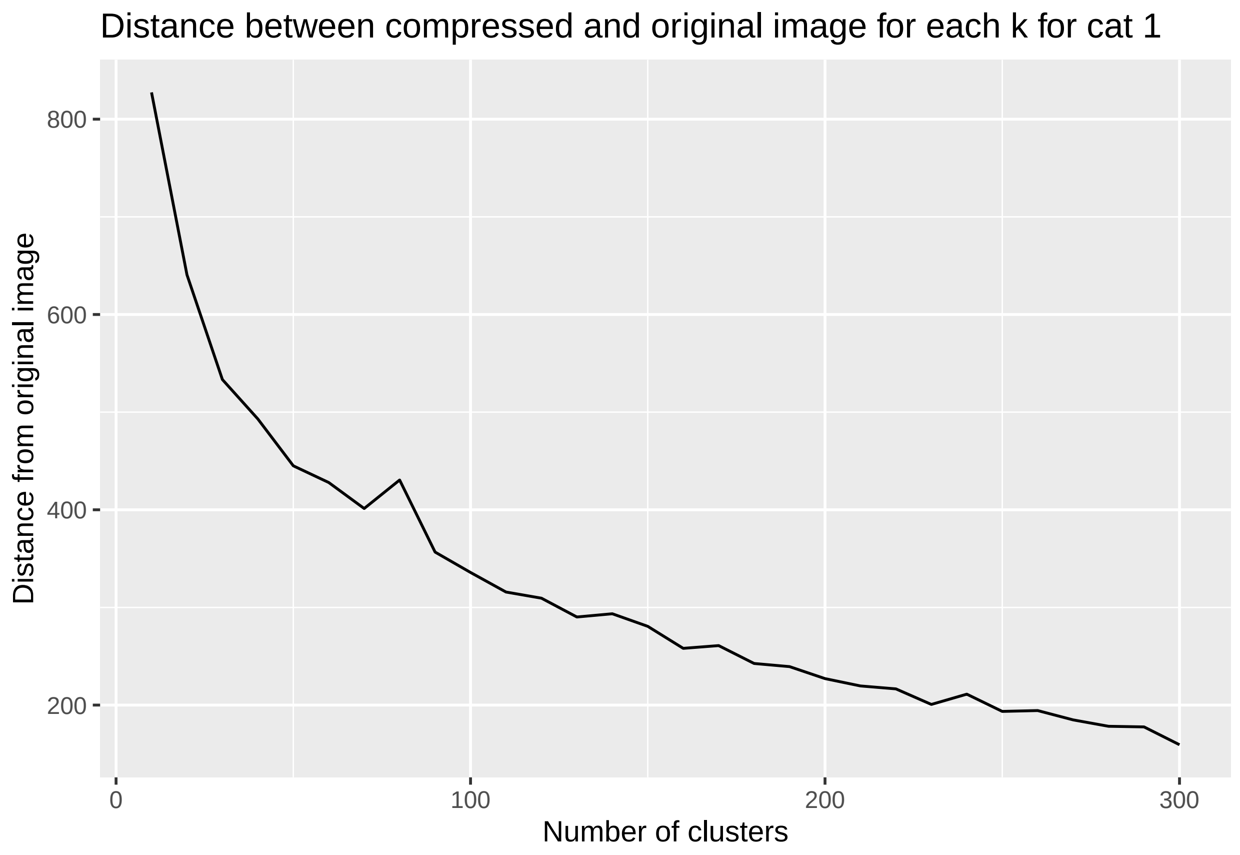
a. Creating the function to evaluate the distance for $k = 10, 20, ..., 300$ clusters

From here, we can make a function that first receive the original image row, then for each value $k = 10, 20, ..., 300$, we will compress the image, reconstructed the compressed image as a squared matrix, and then calculate the distance of that image and the original image. From here, we can make a dataframe with 1 column to be the value k and another column would be the distance of compressed image from the original image. With all this, we can make a line plot and use that with the elbow method to find the optimal number of k for that image. And we implemented this in R with the find_optimal_k function:

```r
find_optimal_k <- function(row_image, n, cat_index) {
  # first we need to create the dataframe that store the result
  result <- data.frame(k = seq(10, 300, by = 10),
                       dist_from_original = numeric(length = 30))
  # first we need to construct the original image
  original_image <- matrix(as.numeric(row_image),nrow=n,byrow=T)
  # Now for each value of k, we create the compressed image as the kmeans object
  # then we remake the kmeans object into a matrix
  # and then we calculate the distance and input into the dataframe
  for (i in 1:30) {
    curr_k <- result[i, 1]
    compressed_image <- compress_image(row_image, n, curr_k)
    result[i, 2] <- image_dist(original_image, compressed_image)
  }
  p <- ggplot(data = result, aes(x = k, y = dist_from_original)) +
    geom_line() +
    labs(x = "Number of clusters",
         y = "Distance from original image",
         title = paste("Distance between compressed and original image for each k for cat",
                       as.character(cat_index))) # we need cat_index only for the title
  print(p)
}
```

We can test to see if this works by looking at the plot for cat image 1

```r
set.seed(12345)
find_optimal_k(image_matrix[1, ], 64, 1)
```

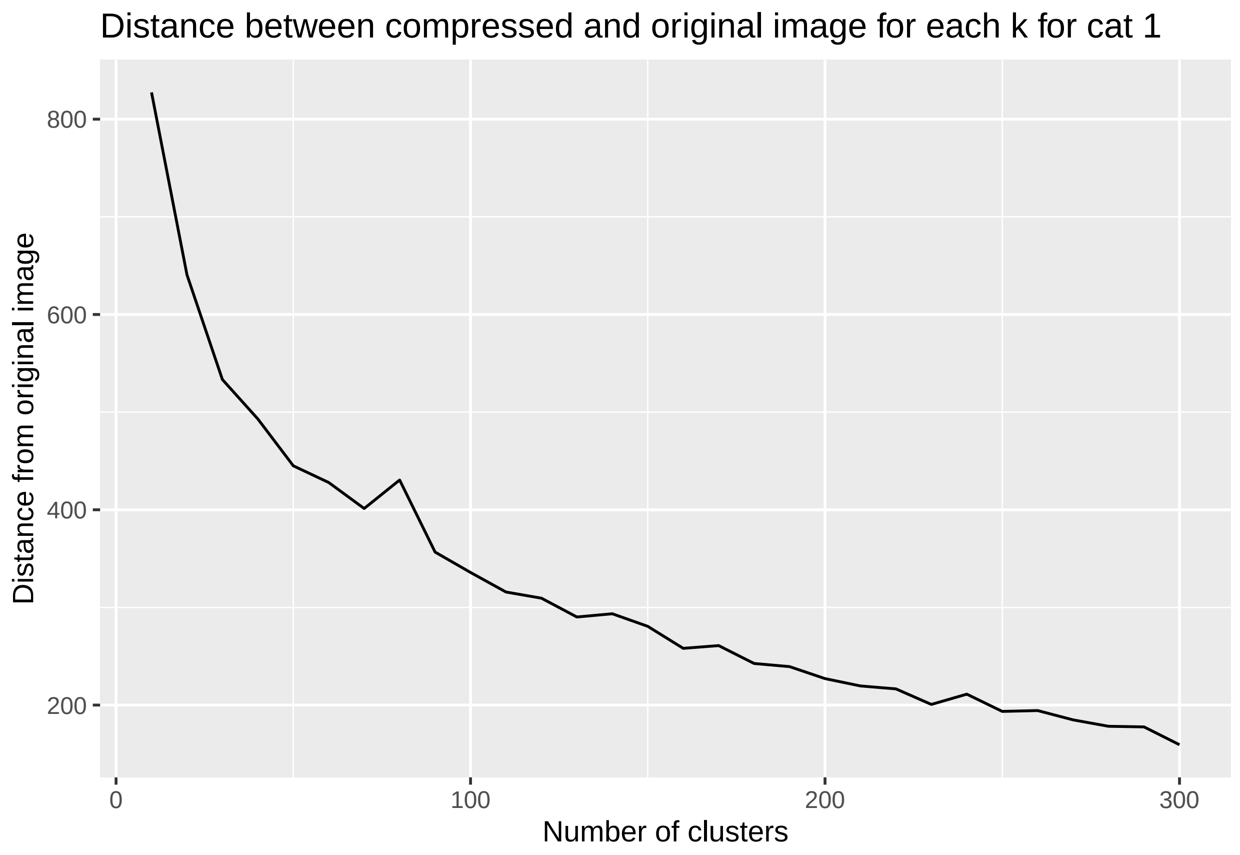


b. Find the optimal number of k for some cats

From here, we can try to replicate the same thing for some cats to see what is the optimal value for k.

Cat number 1

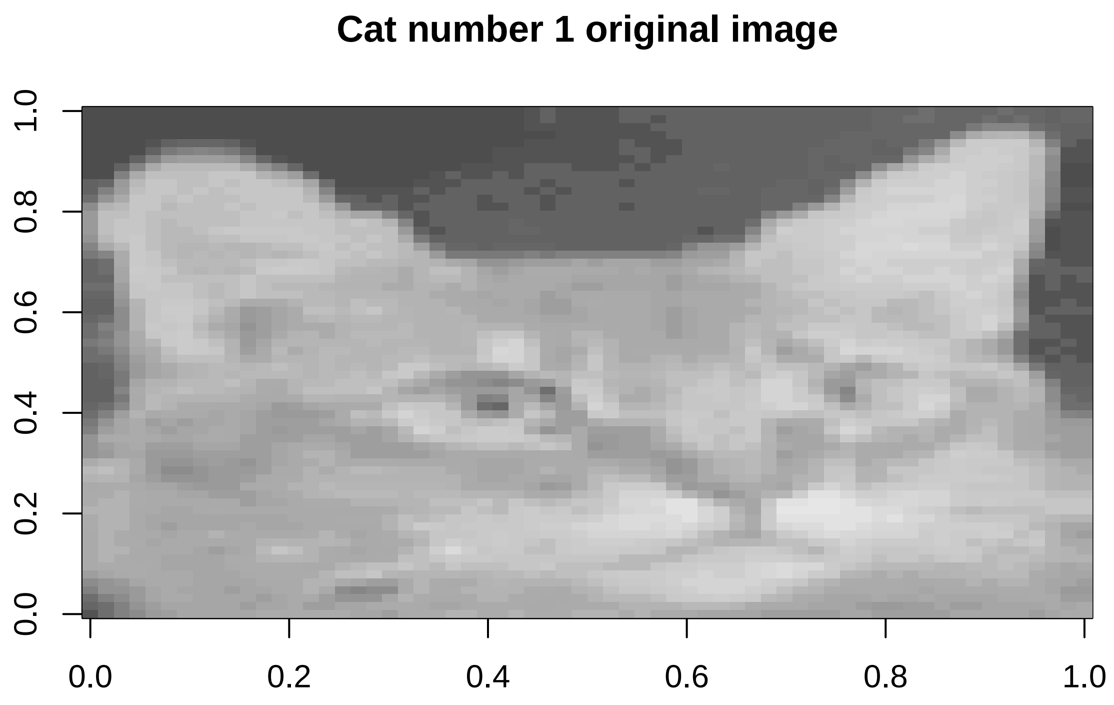
```
set.seed(12345)
find_optimal_k(image_matrix[1, ], 64, 1)
```



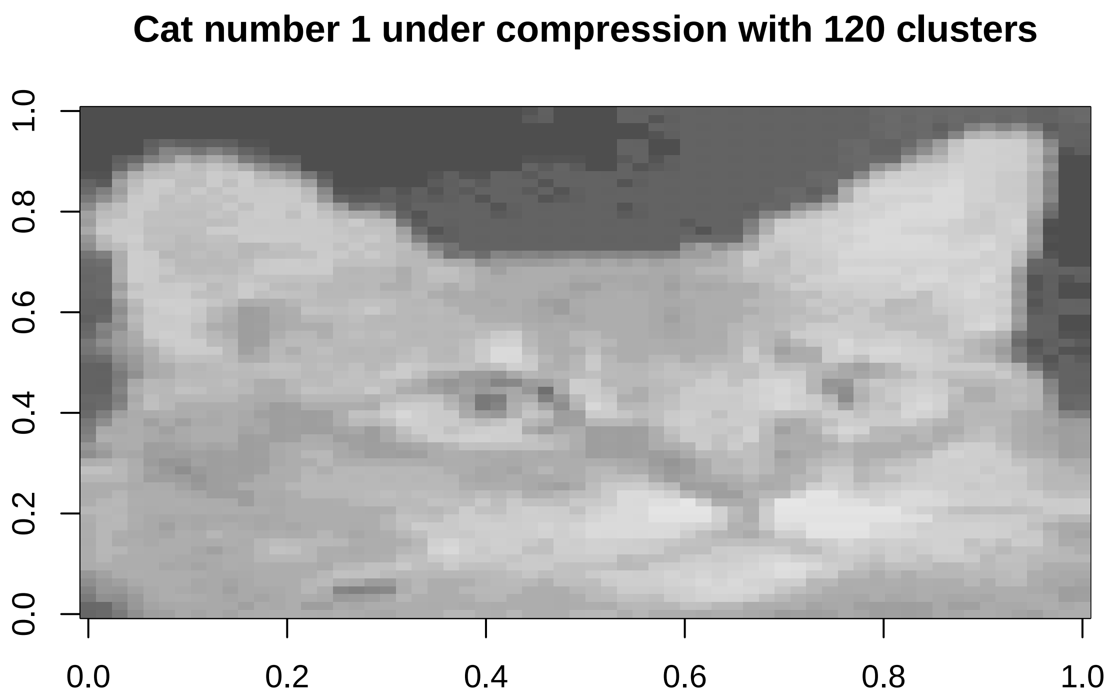
Firstly, as expected with higher value of k the distance between the original image and compressed image decreases. We can use the elboy plot logic and say that for $k = 120$ the image is well compressed and it is not that "far" from the original image, for higher values of k the image does not seem to decrese in distance as much.

```
set.seed(12345)

plotImage(image_matrix[1,])
title("Cat number 1 original image")
```

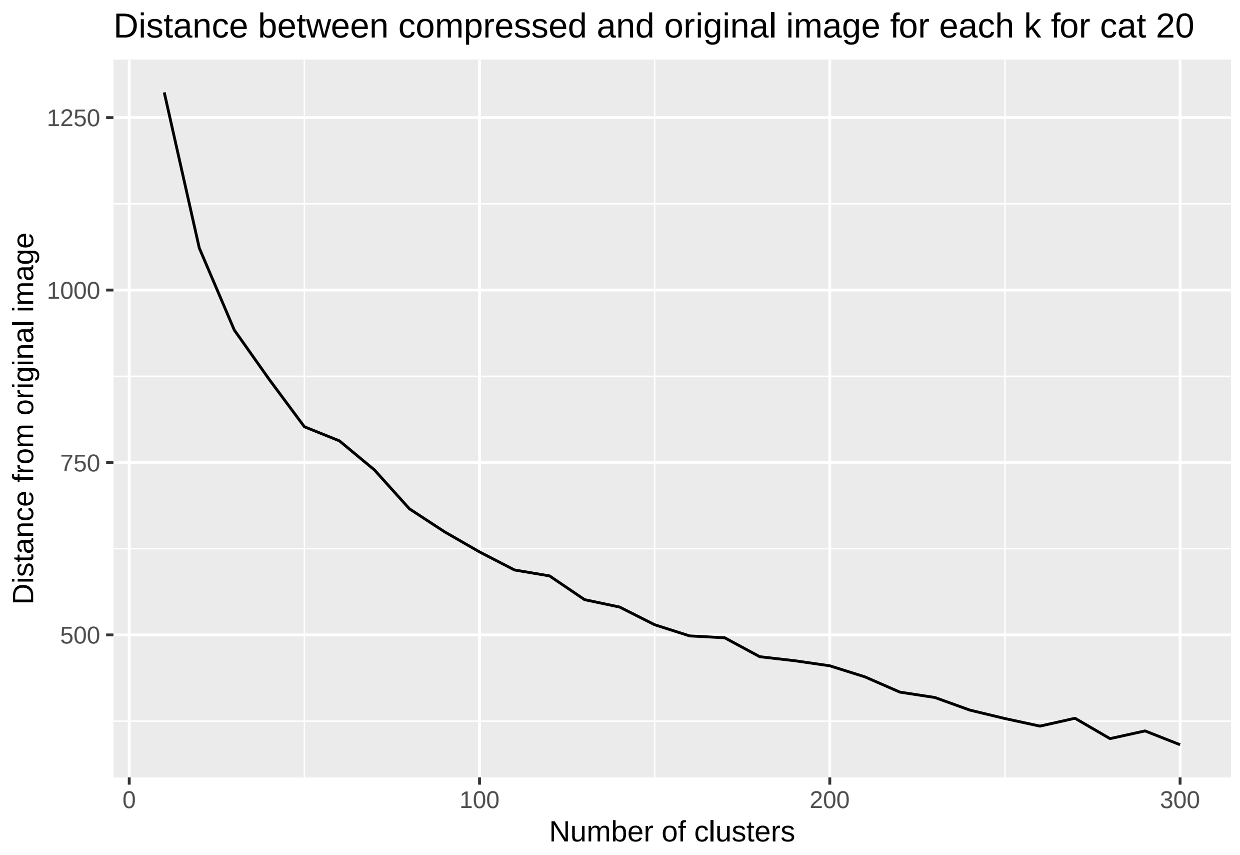


```
plotImage(t(compress_image(image_matrix[1,], 64, 120)))
title("Cat number 1 under compression with 120 clusters")
```



Cat number 20

```
set.seed(12345)
find_optimal_k(image_matrix[20, ], 64, 20)
```

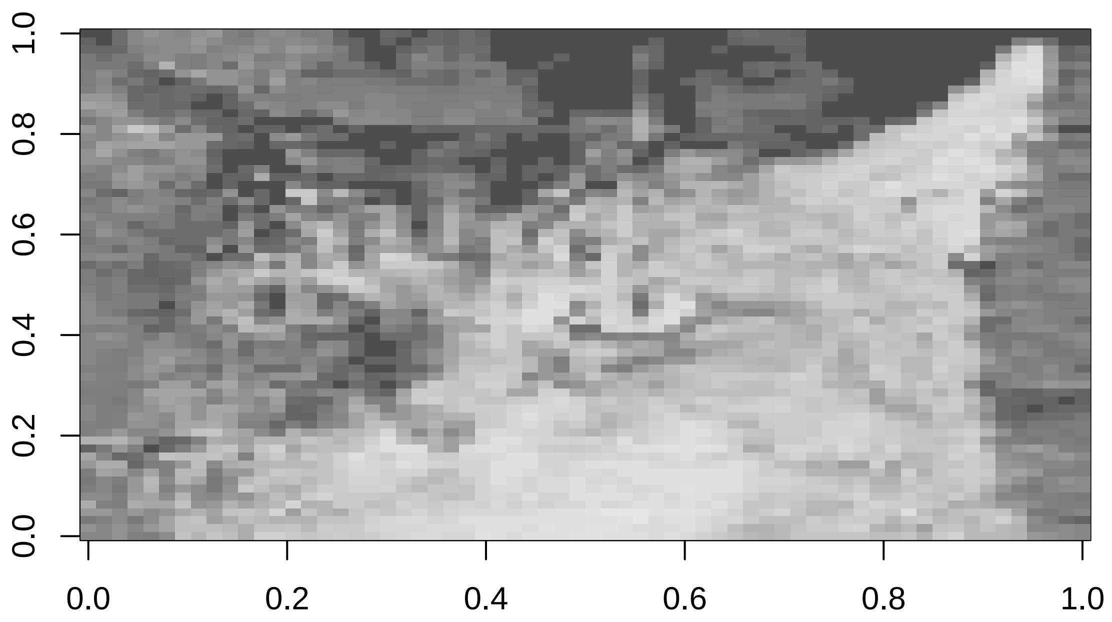


For cat number 20 we can see that starting from value $k = 150$ the distance does not decrease as much anymore, so we can consider that as optimal value of k.

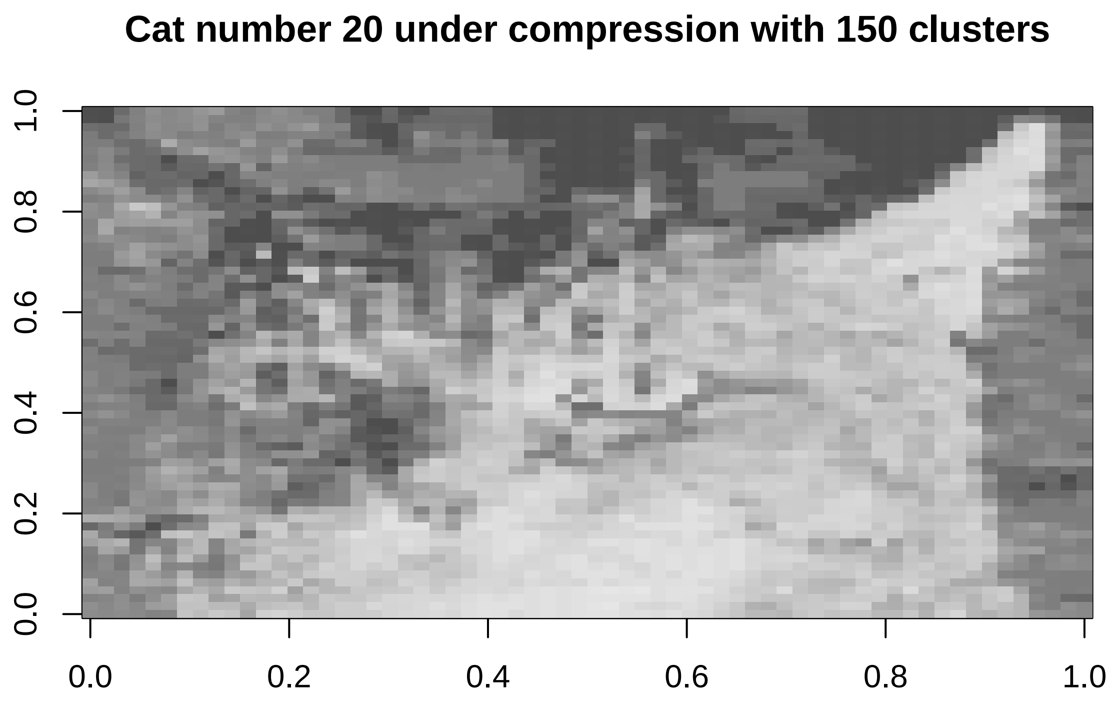
```
set.seed(12345)

plotImage(image_matrix[20,])
title("Cat number 20 original image")
```

Cat number 20 original image

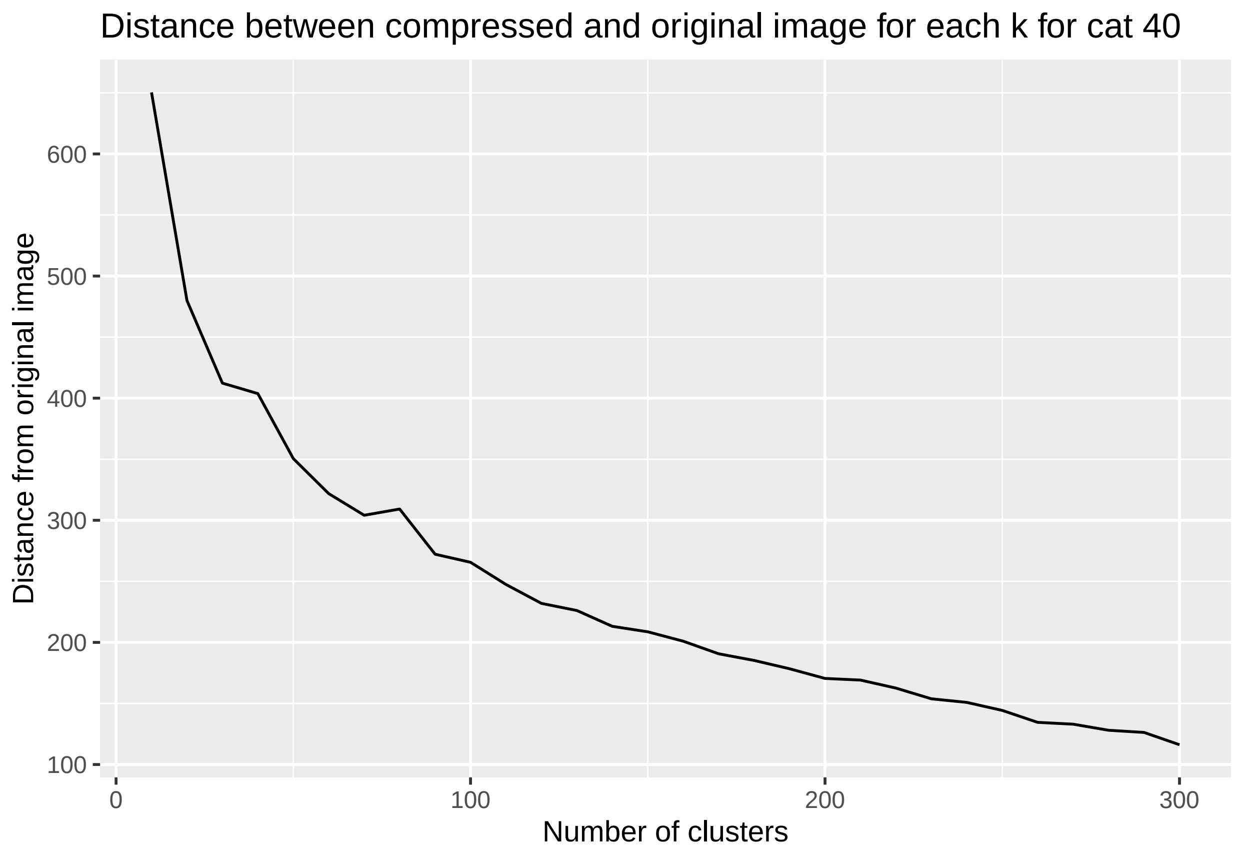


```
plotImage(t(compress_image(image_matrix[20,], 64, 150)))
title("Cat number 20 under compression with 150 clusters")
```



Cat number 40

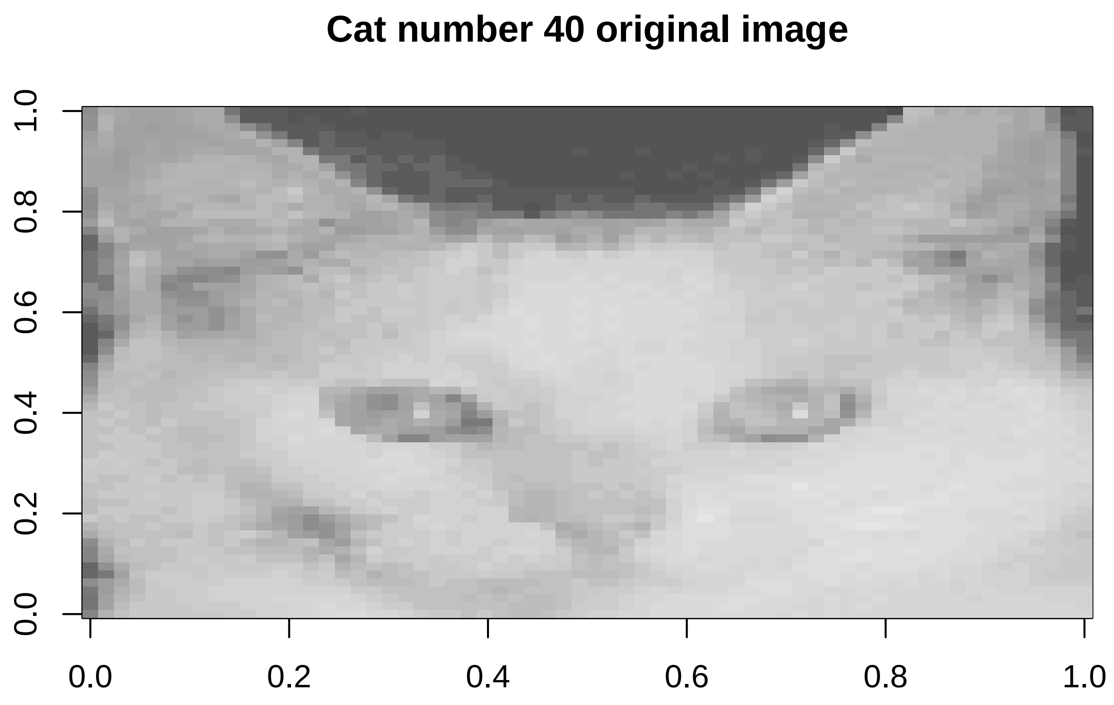
```
set.seed(12345)
find_optimal_k(image_matrix[40, ], 64, 40)
```



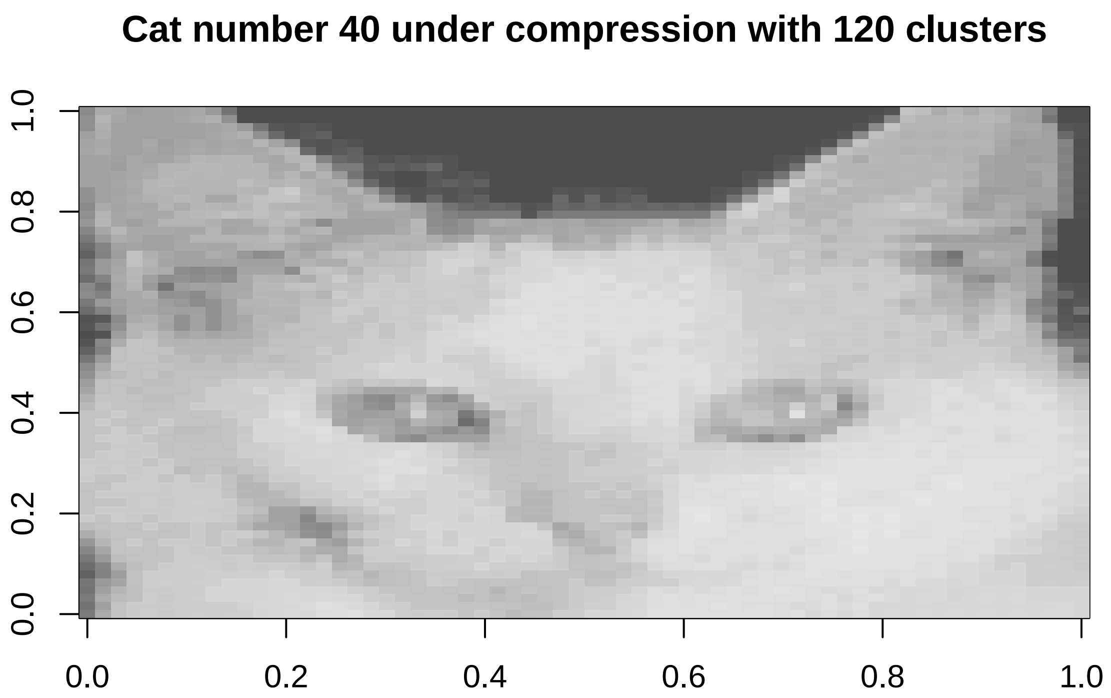
Similar to the first cat starting from value of $k = 120$ the distance does not decrease noticeably so we can consider it the optimal k.

```
set.seed(12345)

plotImage(image_matrix[40,])
title("Cat number 40 original image")
```

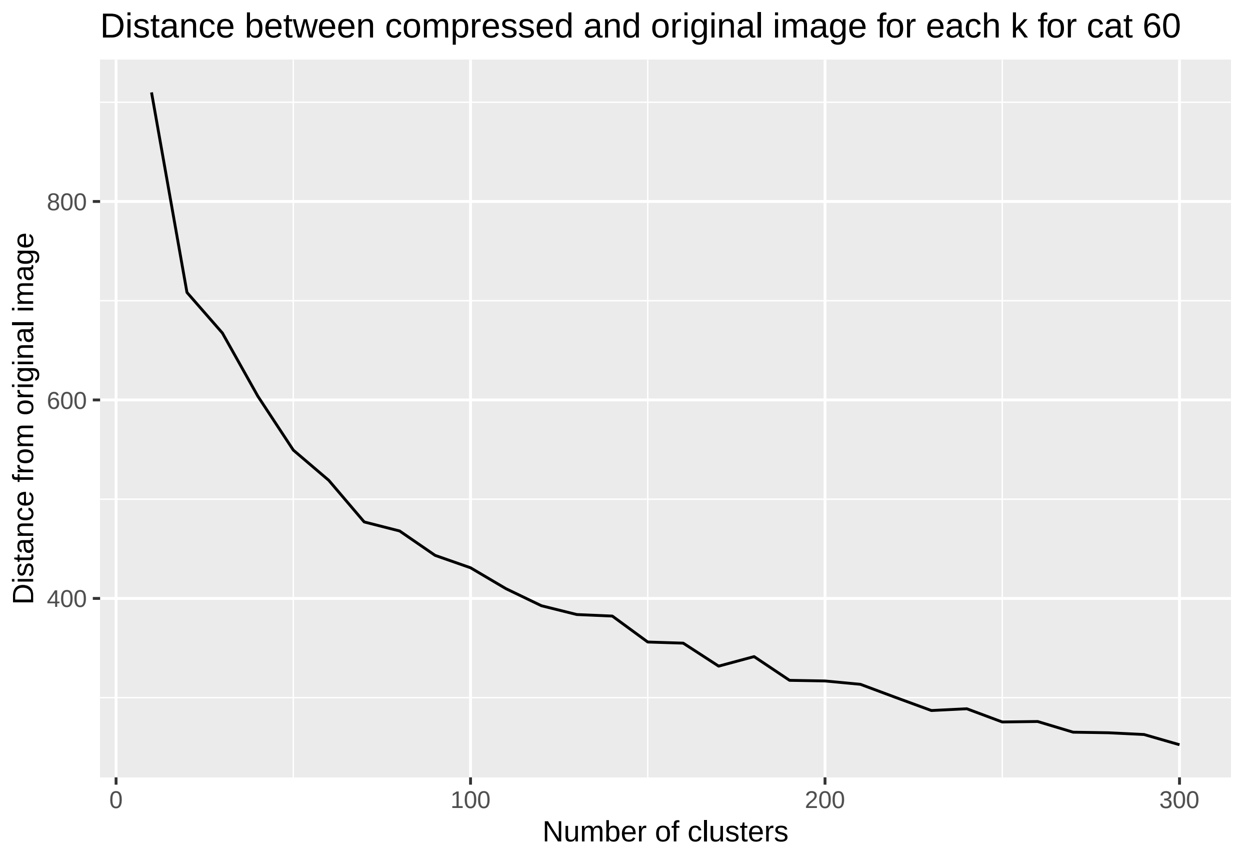


```
plotImage(t(compress_image(image_matrix[40,], 64, 120)))
title("Cat number 40 under compression with 120 clusters")
```



Cat number 60

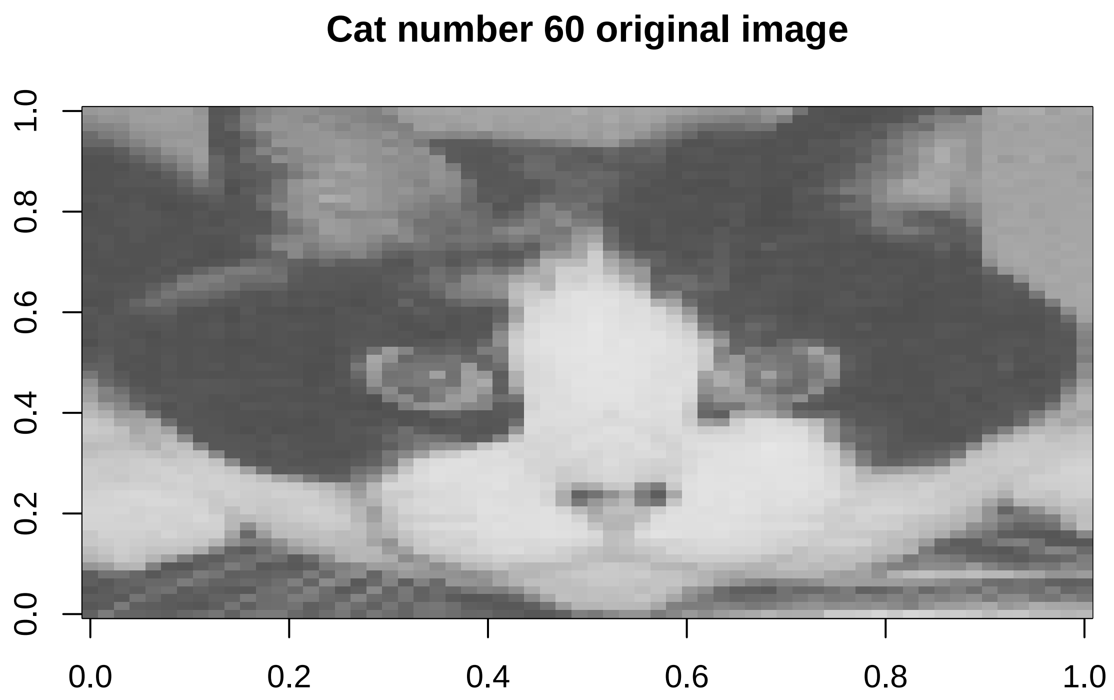
```
set.seed(12345)
find_optimal_k(image_matrix[60, ], 64, 60)
```



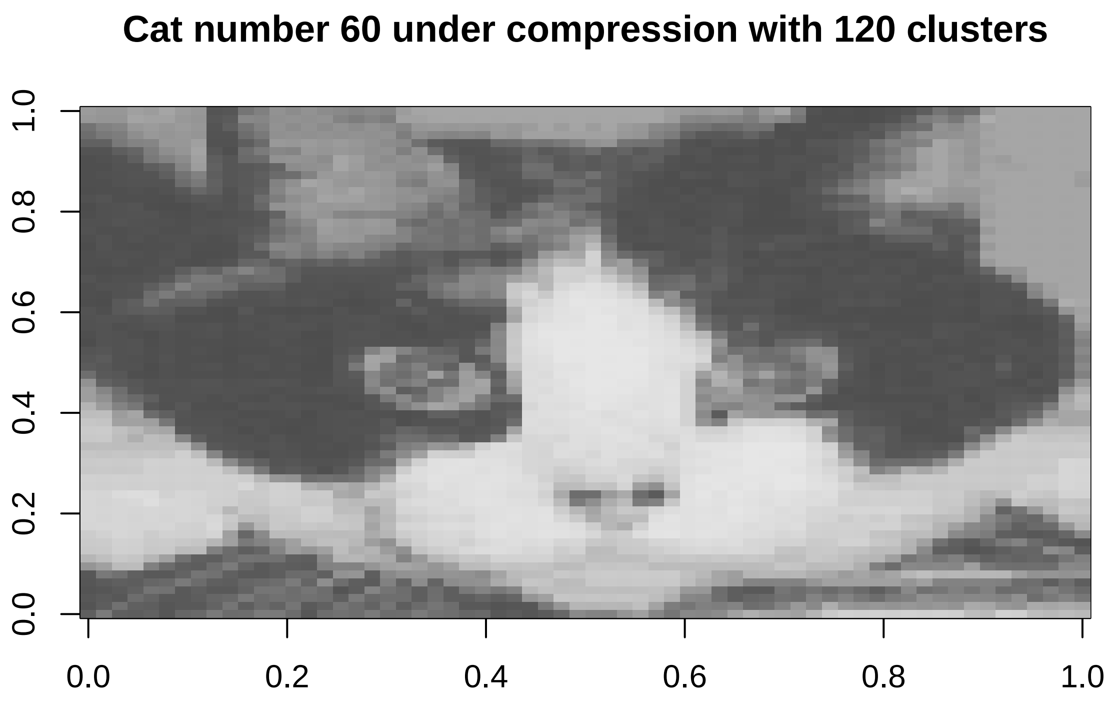
Similar to the first cat starting from value of $k = 120$ the distance does not decrease noticeably so we can consider it the optimal k.

```
set.seed(12345)

plotImage(image_matrix[60,])
title("Cat number 60 original image")
```

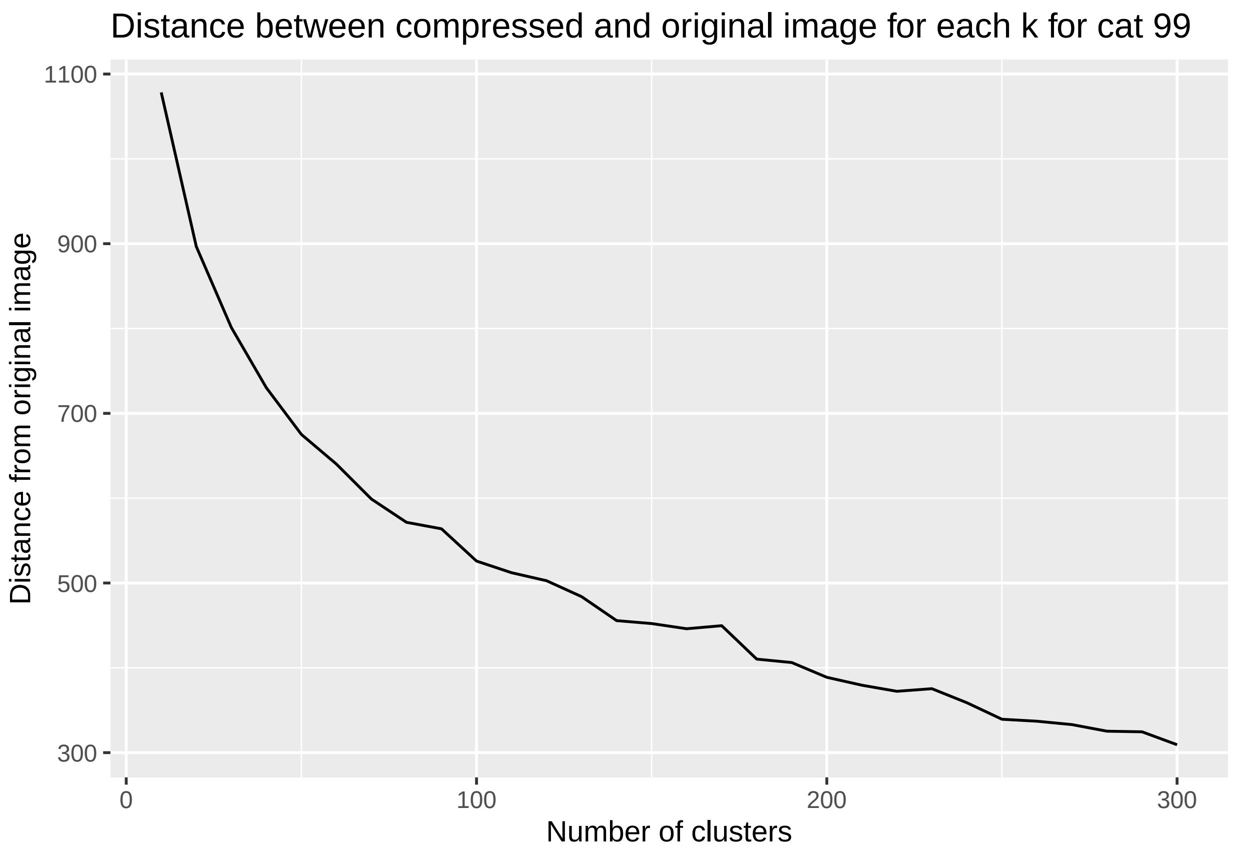


```
plotImage(t(compress_image(image_matrix[60,], 64, 120)))
title("Cat number 60 under compression with 120 clusters")
```



Cat number 99

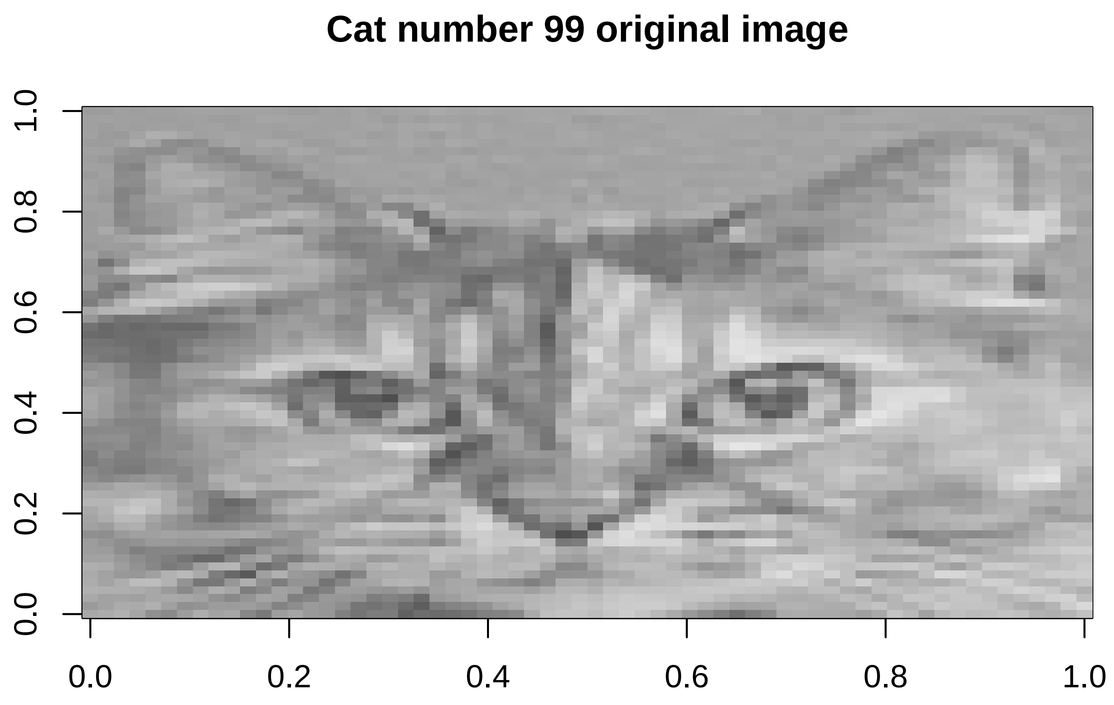
```
set.seed(12345)
find_optimal_k(image_matrix[99, ], 64, 99)
```



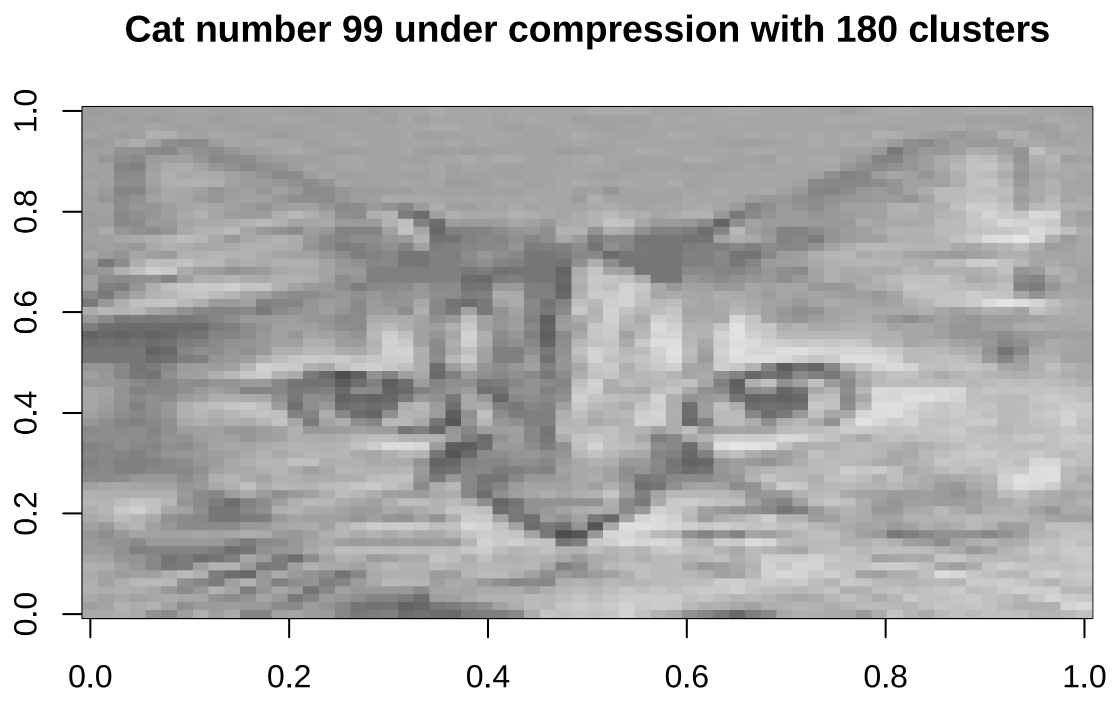
For cat number 99 we can see that starting from value $k = 180$ the distance does not decrease as much anymore, so we can consider that as optimal value of k.

```r
set.seed(12345)

plotImage(image_matrix[99,])
title("Cat number 99 original image")
```



```
plotImage(t(compress_image(image_matrix[99,], 64, 180)))
title("Cat number 99 under compression with 180 clusters")
```



Conclusion

To conclude, we successfully were able to define function compress_image, that given an image and rate of compression returns a matrix of pixels for the compressed image that can be plotted. Using that we also defined a function to compare two given images which helped us to apply compression on different images for a range of values of k and determine an optimal rate of compression. For the five cat images that we tried on average for $k = 120$ there was a good compression rate while the images did not look too different from the original image.