Unsupervised Machine Learning 2023

Tom Maullin

2023-11-14

Introduction to Unsupervised Learning

Welcome to the Unsupervised Learning tutorial 2023.

In the next part of today's tutorial, we are going to used unsupervised machine learning methods to investigate one of the most famous datasets used for machine learning; the MNIST (Modified National Institute of Standards and Technology) database. The full dataset consists of 70,000 training images of hand-written digits between 0 and 9. Today we shall only use a sample of 1,500 of these digits.

As with the supervised learning tutorial, if you have any questions throughout the practical, please feel free to ask any of the in-person tutors, or alternatively, you can email Tom Maullin at tommaullin@gmail.com. When working through this R markdown notebook, please ensure you try all of the `write your code here` blocks and please do not use AI assistance, such as ChatGPT or GitHub Copilot, to assist you coding.

The MNIST dataset

Before we get started, anyone who has come across the MNIST dataset before will know that alongside the 70,000 images, labels are provided for the first 60,000. As a result, this dataset is often used to test supervised learning methods, rather than unsupervised. You may be wondering; why are we using MNIST here instead of in the supervised learning practical?

The answer to this question is that, at the end of this tutorial you will find a challenge. The challenge invites you to use everything you have learned over the this practical and the last to try and obtain a high prediction accuracy on the MNIST dataset via whichever machine learning methods you prefer. The previous practical taught you some supervised methods which you may find useful, and this practical shall help you get to grips with the MNIST data.

To begin, let's load in the dataset.

```
# Load the dataset
load("digit_img.RData")
```

We shall also load in some support functions which will be useful for working with this dataset.

```
source("DigitsUtils.R")
```

You will now find that your workspace contains a variable named `digit_img`.

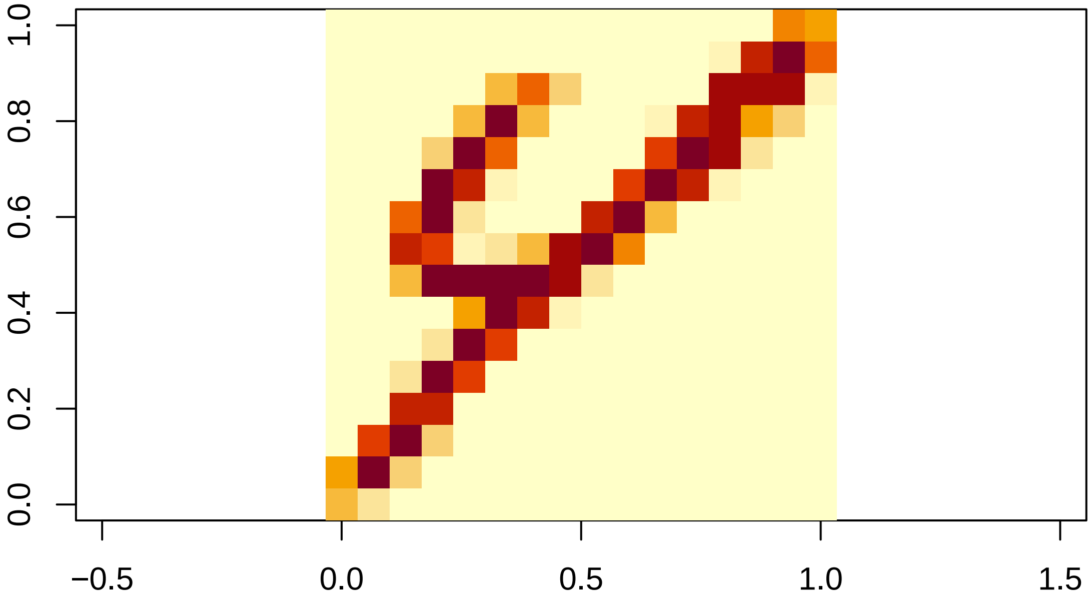
`digit_img` is a 256×1500 matrix, where the first dimension is "space" unwrapped (i.e. the images of handwritten digits are $16 \times 16 = 256$ arrays of pixels), and the second dimension are the 1500 cases. We shall treat the first 1000 as your training data and the remaining 500 as test data.

```
# Get training data
training_data <- t(digit_img[, 1:1000])

# Get testing data
testing_data <- t(digit_img[, 1001:1500])
```

Explore the data. Plotting a single case `plot(digit_img[,1])` won't be so interesting (try it!), as it will just show the "unwrapped" images. Reshape the vector into a matrix and view with the image display function.

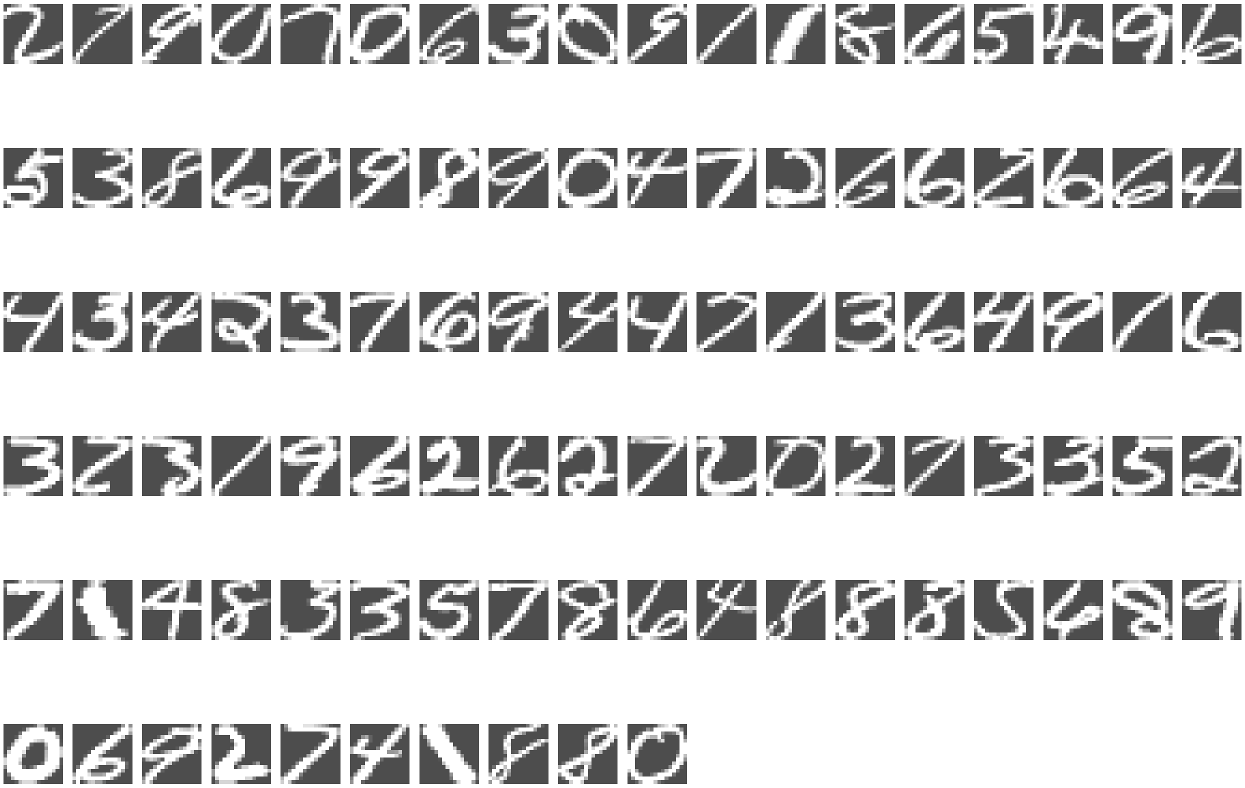
Try running this chunk several times…

```
# Choose a random digit
i = sample(1:1500,1)

# Reshape the ith datapoint into an image
img_matrix = matrix(digit_img[,i], nrow=16)

# Flip the image upside down (it will appear upside down otherwise)
flipped_matrix = img_matrix[nrow(img_matrix):1, ]

# Show the final image
image(t(flipped_matrix), asp=1)
```



Getting images to look right is a bit of a pain, so you can use the `display_digit` function instead. For example, to see the first 100 digits…

```
display_digit(digit_img[,1:100])
```

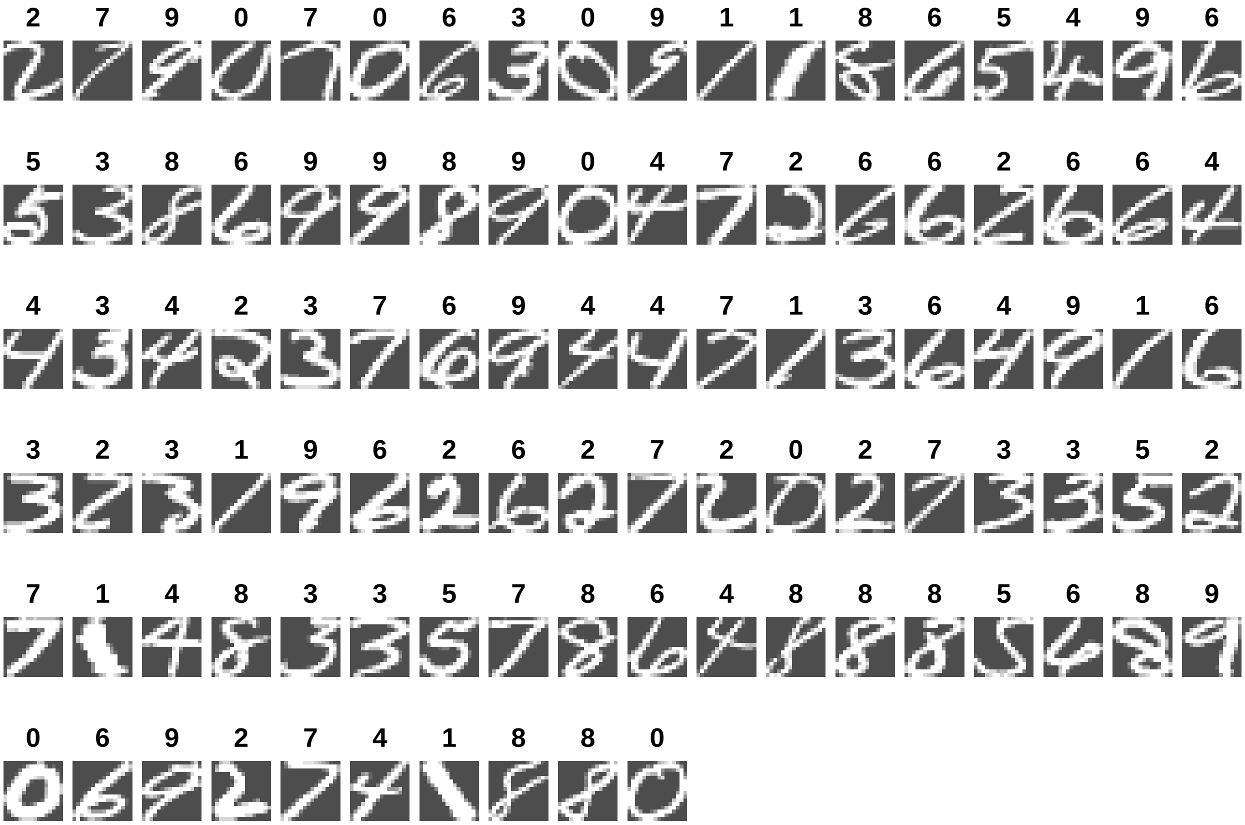


Although we shall not use the labels in this dataset for building models just yet, we shall load them into memory now. We will use these to help interpret the machine learning procedures' behavior. The `digit_lab` variable is a length 1500 label vector telling you the label (true identity) of the digit, one of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (10 means "zero", so we convert it below).

```
# Load the labels
load("digit_lab.RData")

# Replace elements equal to 10 with 0 (it's easier to view this way)
digit_lab[digit_lab == 10] <- 0
```

You can view the labels alongside the images using the `display_digit` function as follows:

```
display_digit(digit_img[,1:100],digit_lab[,1:100])
```



Let's split the labels into training and testing.

```
# Get training labels
training_labels <- t(digit_lab[, 1:1000])

# Get testing labels
testing_labels <- t(digit_lab[, 1001:1500])
```

K-means clustering

We're now going to perform K-means clustering on this dataset. Our aim here is to investigate whether or not `kmeans` will provide a clustering that reflects the digits written in the images (e.g. will it group all the images of fours together, for instance). To achieve this, we will use the `kmeans` function, which can be found in the R `stats` package. The below code will apply k-means on our training data.

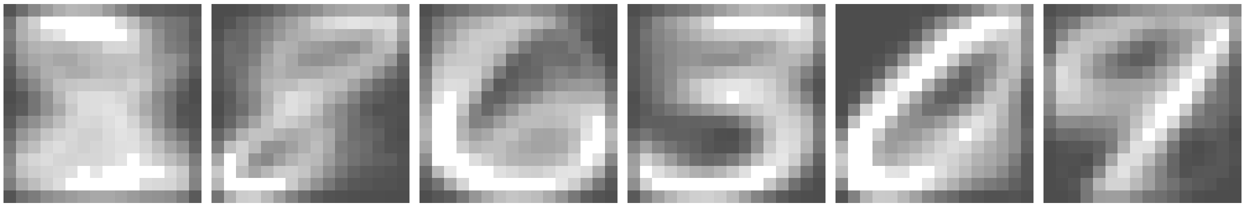
```
# Choose a k for k-means
k = 10

# Applying k-means clustering
set.seed(123)  # Setting a seed for reproducibility
clusters <- kmeans(training_data, centers=k)
```

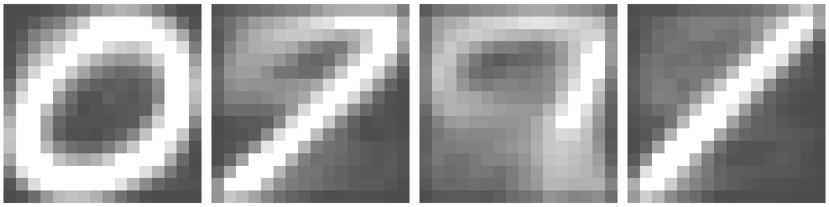
Have a look in the `clusters` object. What do you think the different variables refer to?

The below code will now create images of the `cluster$centers`.

Question: What do you think the `cluster$centers` are? What might these images correspond to? Do they resemble the digits? Why/why not?

```r
# Display the cluster centers
display_digit(t(clusters$centers))
```





Viewing the cluster centers is interesting, but not very informative on it's own. The below code will loop through the clusters for you, count the number of datapoints (images) in each cluster and work out which label is the most frequent for that cluster. It will also report how many images in that cluster had that label. For example, in a cluster consisting of 10 images, eight of which are images of the number 7, the number of elements is 10, the most frequent element is 7, and it occurs 80% of the time.

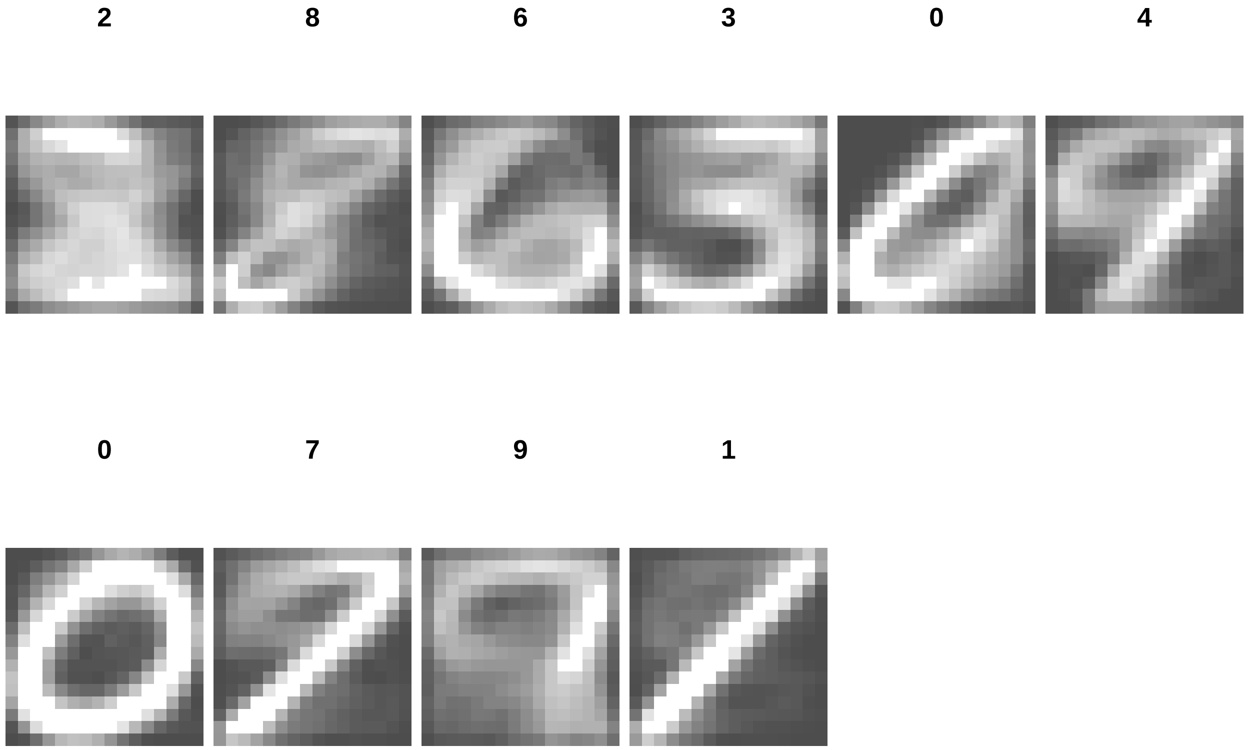
```r
# Initialize vectors to store the results
cluster_labels <- numeric(k)
most_frequent_labels <- numeric(k)
frequencies <- numeric(k)
num_elements_in_cluster <- numeric(k)

# Loop through the clusters
for (j in 1:k){

  # Get the labels for the clusters in the images
  selected_labels <- training_labels[clusters$cluster == j]

  # Calculate the frequency of each element
  label_frequencies <- table(selected_labels)

  # Sort the frequencies in decreasing order
  sorted_frequencies <- sort(label_frequencies, decreasing = TRUE)
  # Extract the count of the most frequent element
  most_frequent_count <- as.numeric(sorted_frequencies[1])

  # Get the label value of the most frequent element
  most_frequent_label <- names(sorted_frequencies)[1]

  # Calculate the total number of elements in selected_labels
  total_count <- length(selected_labels)

  # Calculate the percentage
  percentage_most_frequent <- (most_frequent_count / total_count) * 100

  # Store the results in the vectors
  cluster_labels[j] <- j
  most_frequent_labels[j] <- most_frequent_label
  frequencies[j] <- percentage_most_frequent
  num_elements_in_cluster[j] <- total_count

}

# Construct the dataframe
cluster_info <- data.frame(
  cluster_number = cluster_labels,
  size_of_cluster = num_elements_in_cluster,
  most_frequent_label = most_frequent_labels,
  frequency = frequencies
)

# Print the dataframe
print(cluster_info)
```

```
##    cluster_number size_of_cluster most_frequent_label frequency
## 1               1              77                   2  48.05195
## 2               2             130                   8  46.92308
## 3               3              73                   6  69.86301
## 4               4             147                   3  57.82313
## 5               5              94                   0  41.48936
## 6               6              88                   4  51.13636
## 7               7              53                   0  92.45283
## 8               8             119                   7  39.49580
## 9               9             101                   9  31.68317
## 10             10             118                   1  62.71186
```

> **Question:** What do you think of your results? Is the method correctly clustering different digits? Why/why not?

For reference, here is an image of the clutser centers, each with their corresponding most frequent label. Do these images look like the labels?

```r
display_digit(t(clusters$centers), cluster_info$most_frequent_label)
```

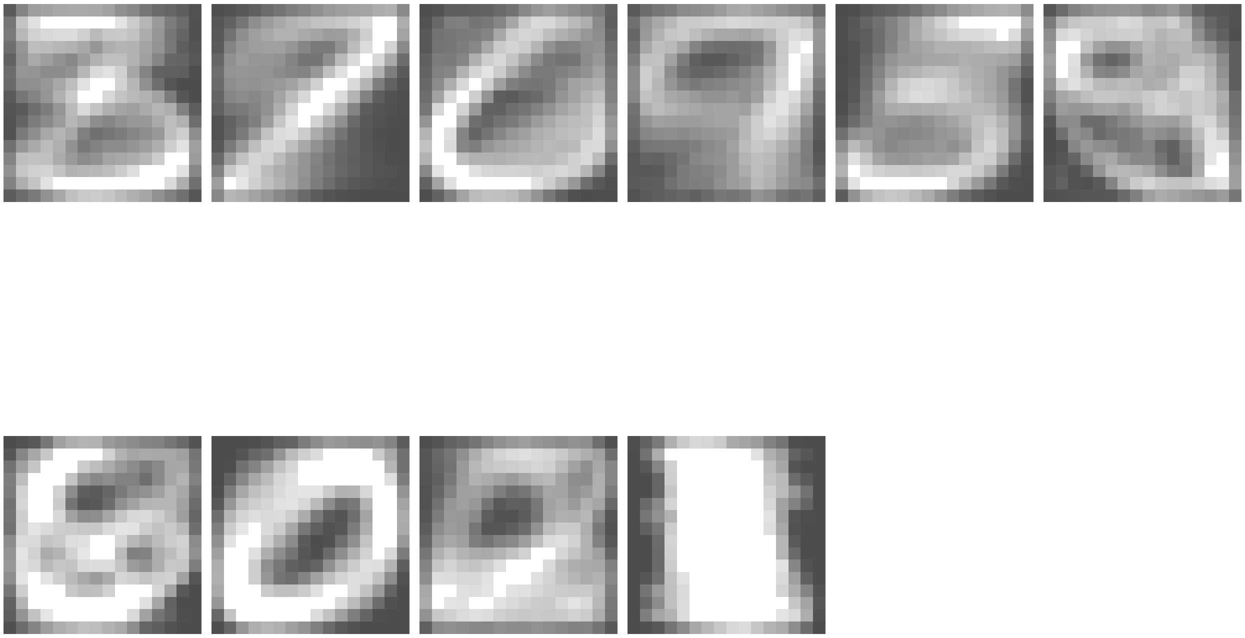


> **Question:** Try rerunning the above code with a higher value of k (say `k=200`). What do you notice about your clusters? Are the labels more or less consistent within clusters now? Why do you think this is?

Hierachical Clustering

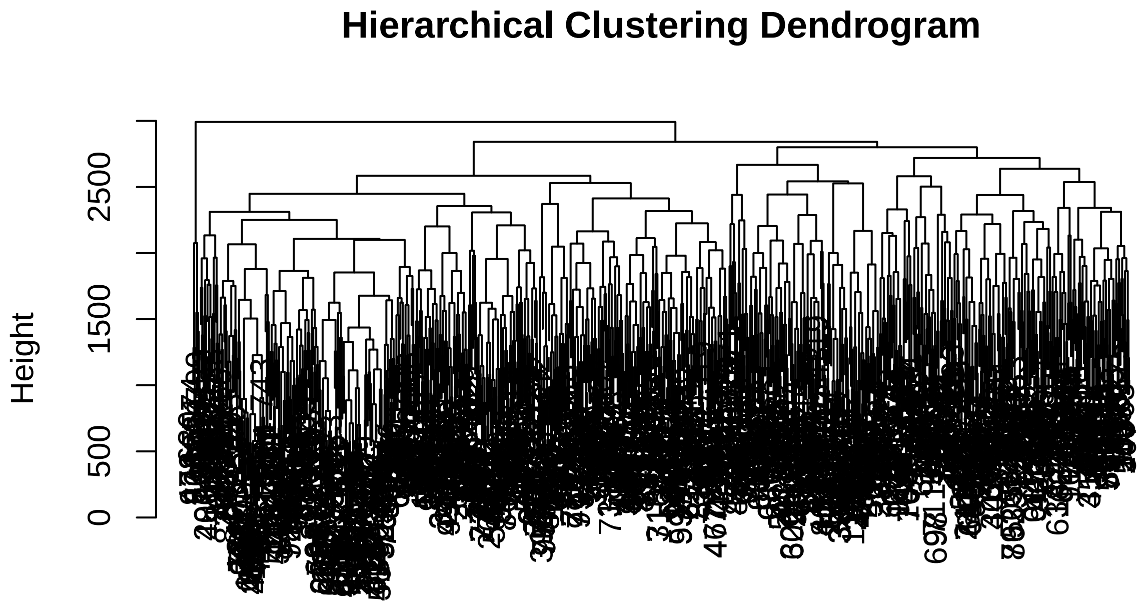
Let's now investigate a different clustering approach. The below code will attempt the same task as above, but using hierarchical clustering. This time we shall use the `hclust` function from the `stats` package.

```r
# Performing hierarchical clustering
hc <- hclust(dist(training_data))

# Cutting the dendrogram to get 10 clusters
cluster <- cutree(hc, k=10)

# Calculating centerss of each cluster
centers <- sapply(1:10, function(k) colMeans(training_data[cluster == k, ]))

# View the results
display_digit(centers)
```



Let's look at the dendrogram for `hc`.

```r
# Plotting the dendrogram
plot(hc, main="Hierarchical Clustering Dendrogram", sub="", xlab="")
```



> **Question:** How should this diagram be interpreted? At which height do you think we cut the dendrogram to perform our clustering?

Based on the code we used for `kmeans`, write your own code below that works out the most frequent label for each cluster and how frequently it appears. As above, add labels to the clusters. Do the labels resemble the images?

> **Question:** Before proceeding, do you think k-means or hierachical clustering will be better at this task? Explain your answer.

```
# Write your code here...
```

Feature Selection

We now move away from clustering to instead focus on a different form of unsupervised learning: Feature selection. Our aim in this section is to see if we can improve the performance of a **supervised** learning algorithm by first applying some **unsupervised** feature selection.

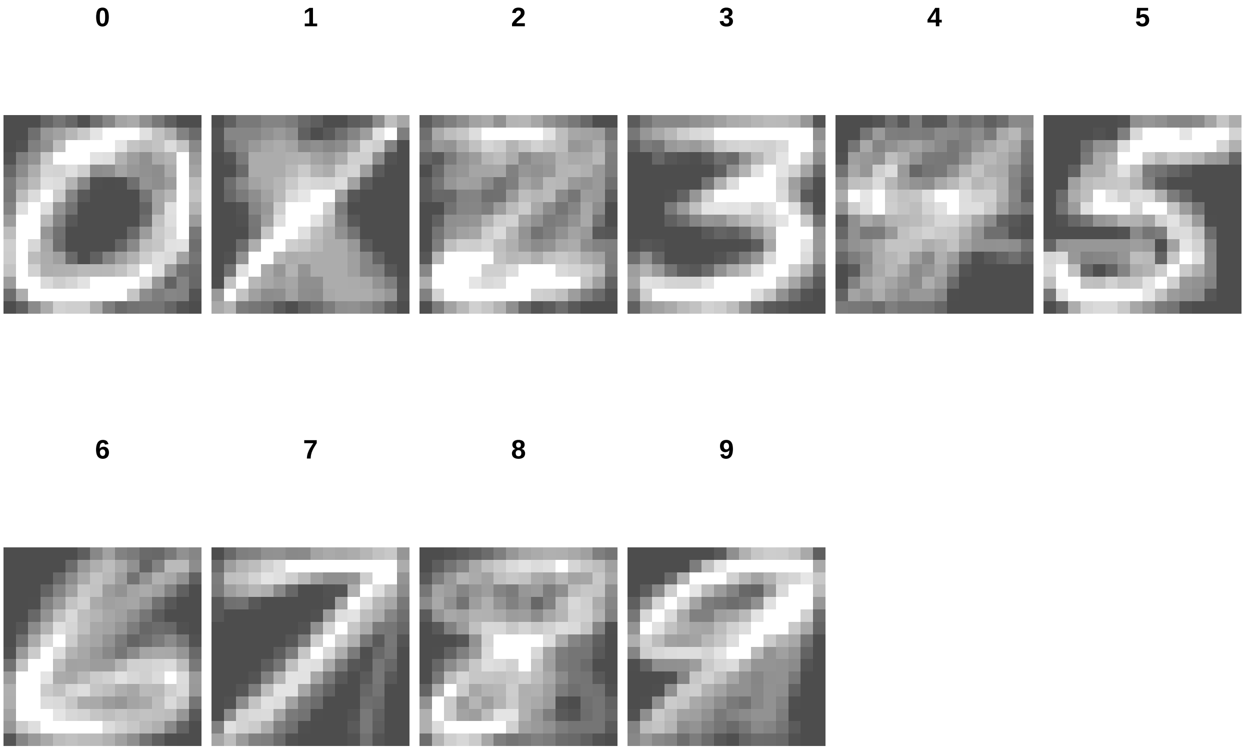
To assist this section, we've provided the function `train_digit`, which takes a matrix (256 × n) and a label vector (length n) and returns the parameters needed to define a Naive Bayesian classifier, i.e. per-digit (class conditional) pixel-wise mean and variance. Open the `DigitsUtils.R` and make sure you understand what it's doing… it's just computing some means and variances.

For example, to train on the first 100 cases

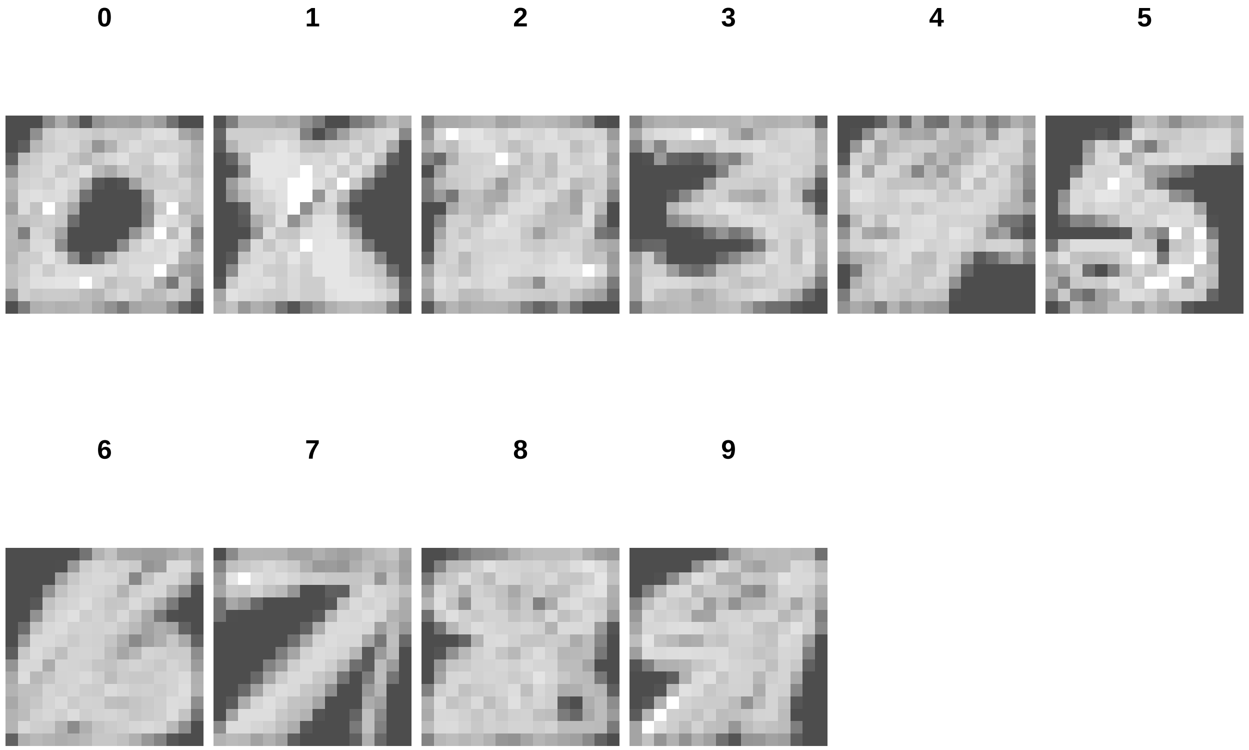
```
I.train=1:100
train = train_digit(digit_img[,I.train],digit_lab[I.train])
```

Be sure to look at these images! Do they make sense?

```r
display_digit(train$ccMean,0:9)        # c.c. mean
```



```r
display_digit(sqrt(train$ccVar),0:9,MaxVal=NA) # c.c. stdev
```



> **Question:** What does this tell you about variability and consistency of how these digits? E.g. compare 1 to 0? Or the variance of 2.

Let's now evaluate your model on some test data. The function `test_digit` in `DigitUtils.R`; takes four arguments:

- Class conditional mean (256 × 10 matrix) – Trained model parameter
- Class conditional variance (256 × 10 matrix) – Trained model parameter
- Features, for N instances (256 × N) – Testing data
- Labels, for corresponding (1 × N) matrix – Testing labels, to compute accuracy

and returns a list with the accuracy and predicted labels.

For example, to evaluate the first 100 test cases, using the first 100 test items, do

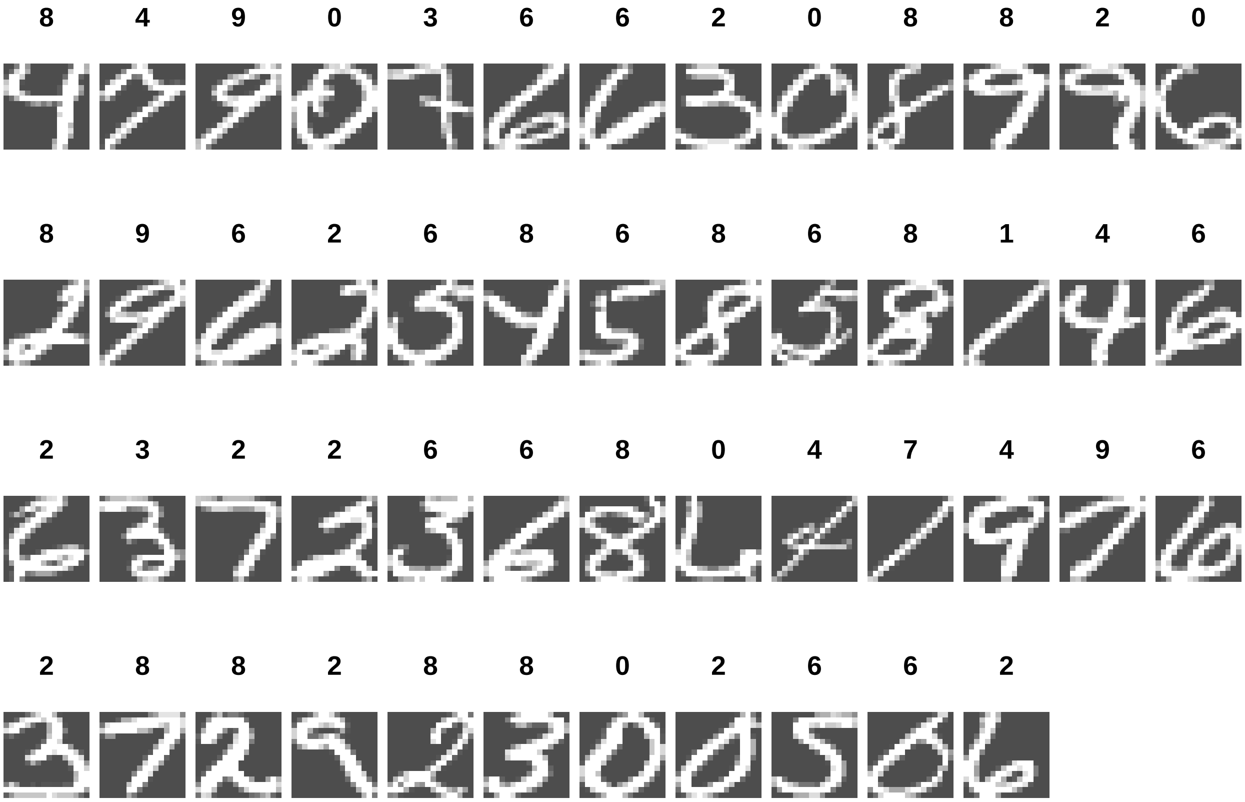
```
eval = test_digit(train$ccMean,train$ccVar, t(testing_data), testing_labels);
eval$Accu
```

```
## [1] 0.468
```

and then `eval$Accu` gives the overall accuracy (fraction correctly classified) and `eval$Pred` is a length 100 vector with the predicted label for each case. Note the labels (true digits) are only used to compute accuracy, and not for prediction.

Examine the predictions with this command:

```
display_digit(t(testing_data)[,1:50],eval$Pred[,1:50])
```



Now repeat this with the full training data (note you will have to use the transposed training data, i.e. `t(training_data)`, in the `train_digit` command). Has the accuracy improved much?

Principal Component Analysis

Now that we have an example analysis to work with, let's see how performing principal component analysis feature extraction helps.

We have provided a function `PCAxform`, which takes a set of training and test data, and transforms it to PCA space using directions determined only the training data. The transformed images won't be an image (it will have <256 rows), but it can be submitted to `digit_train` and `digit_test` just as before. For example, with 20 PCA directions:

```
# Choose 20 PCA directions
K=20

# Transform the data
transformed_data = PCAxform(t(training_data),t(testing_data),K)

# Get the dimension reduced training and testing data
transformed_train=transformed_data[[1]]
transformed_test=transformed_data[[2]]

print(dim(transformed_train))
```

```
## [1]   20 1000
```

Make sure you understand what has happened before proceeding. We now perform our training and testing exactly as before, but using the dimension reduced data.

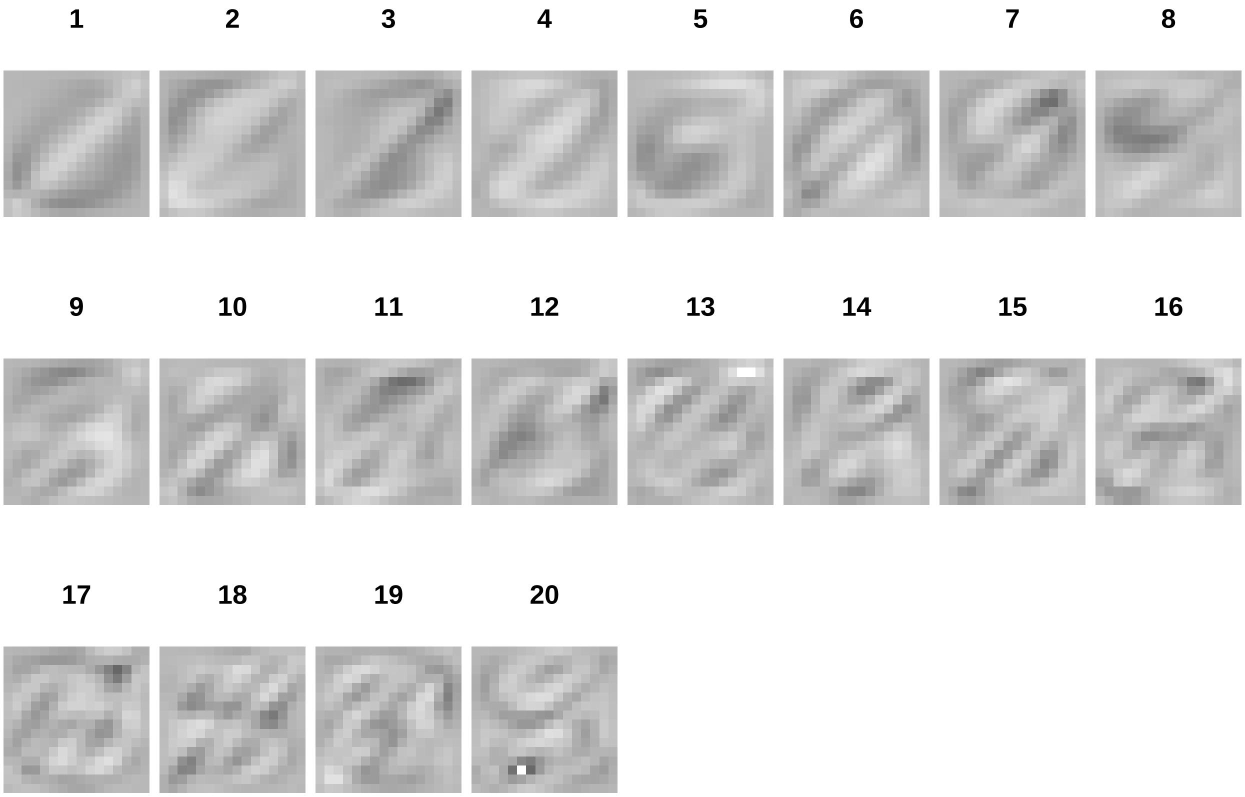
```r
# Perform the niave bayes on the PCA directions, rather than the images themselves.
training_variables = train_digit(transformed_train,training_labels);

# Testing results
test_results = test_digit(training_variables$ccMean,
                          training_variables$ccVar,
                          transformed_test,
                          testing_labels);
test_results$Accu
```

```
## [1] 0.822
```

Does this improve accuracy?

Note you can visualize the PCA directions with the optional outputs `Utrain` and `Dtrain`, the eigenvectors and eigenvalues of the training data:

```r
# Get the eigenvectors and eigenvalues
eigenvectors=transformed_data[[3]]
eigenvalues=transformed_data[[4]]

# Display the eigenvectors
display_digit(eigenvectors,1:K,NULL,NA)
```



What do you think these images represent?

Challenge

Okay, we are now ready to move onto the final task of today; the challenge! Using everything you have learnt today (and whatever you might find online!), try to obtain a better prediction accuracy for the MNIST dataset. You are free to use whatever methods you like and be as creative as you like. We have held onto another 500 datapoints which will be made available to you during the last 10 minutes of the final tutorial, to validate your model.

```
# Write your code here...
```

Best accuracy on the validation set wins!

Acknowledgements

This practical draws heavily from material prepared by Professor Thomas E. Nichols.