BIOS 7659 Homework 9

Tim Vigers

08 December 2020

# 1. Classification

Import and prepare the data:

load("./data/dataHW9-breastcancer.Rdata")  
train = breastcancer[[1]] #training expression data  
trainclass = breastcancer[[2]] #training classes  
test = newpatients #new expression data  
testclass = trueclasses #new classes

## a. Describe the k-nearest neighbor (KNN) algorithm

The first step in the KNN algorithm is to take each sample that you want to classify (the test set), and calculate the distances between those samples and your labeled training set using your favorite distance metric (Euclidean, 1-correlation, norm, etc.). Then, for each new sample you find the closest training samples. Each of these labeled nearest neighbors “votes” on the classification for the new sample, and that sample is labeled by majority vote (e.g. if 3 of the neighbors are labeled “ER+” and 5 are labeled “ER-”, the new sample will be classified as “ER-”). It’s also possible to weight each neighbor’s vote so that closer training samples have more influence than more distant neighbors.

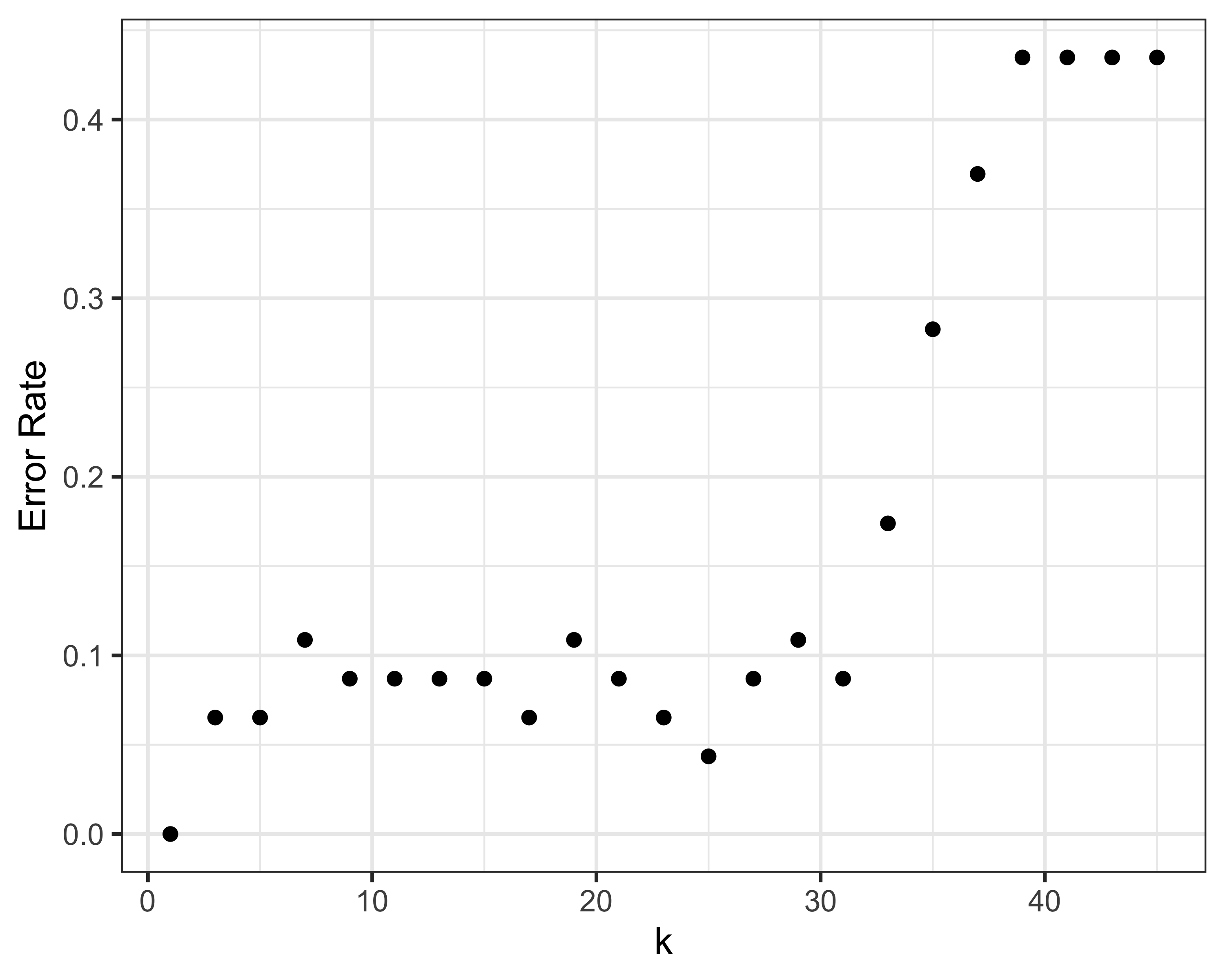
Run the knn() function from the class package:

k = knn(train = t(train),test = t(train),cl = trainclass,k = 3)

Using the train dataset for both training and testing results in 43 correct classifications out of 46 (about 93.5%). It is definitely not good practice to use the same data for training and testing, because it almost guarantees overfitting. Overfitting means that the model does really well on the training data, but not as well on other data, so it has limited real-world application. One way to think about it is that the algorithm is modeling the noise in the data in addition to the signal.

## b. Repeat part a) for multiple values of k.

ks = 1:46  
ks = ks[which(ks%%2==1)]  
e = lapply(ks, function(k){  
 kn = knn(train = t(train),test = t(train),cl = trainclass,k = k)  
 sum(kn != trainclass)/length(trainclass)  
})  
p = as.data.frame(cbind(ks,unlist(e)))  
ggplot(p,aes(x=ks,y=V2)) + geom\_point() + theme\_bw() + xlab("k") + ylab("Error Rate")



I only looked at odd values of here because it’s best to avoid issues with potential ties (especially when there are only two classes), although the knn() function does include some tie-breaking methods that allow for equal values of k. Also, you cannot use a value of k that is greater than the number of observations, so the range of k is .

Based on the plot above, I would use because it produces the lowest error rate (about 4.3%) among the reasonable values for . In this case where we are using the same data for training and testing, we would definitely not want to use , because it is really just matching each point to itself (which explains the 0% error rate).

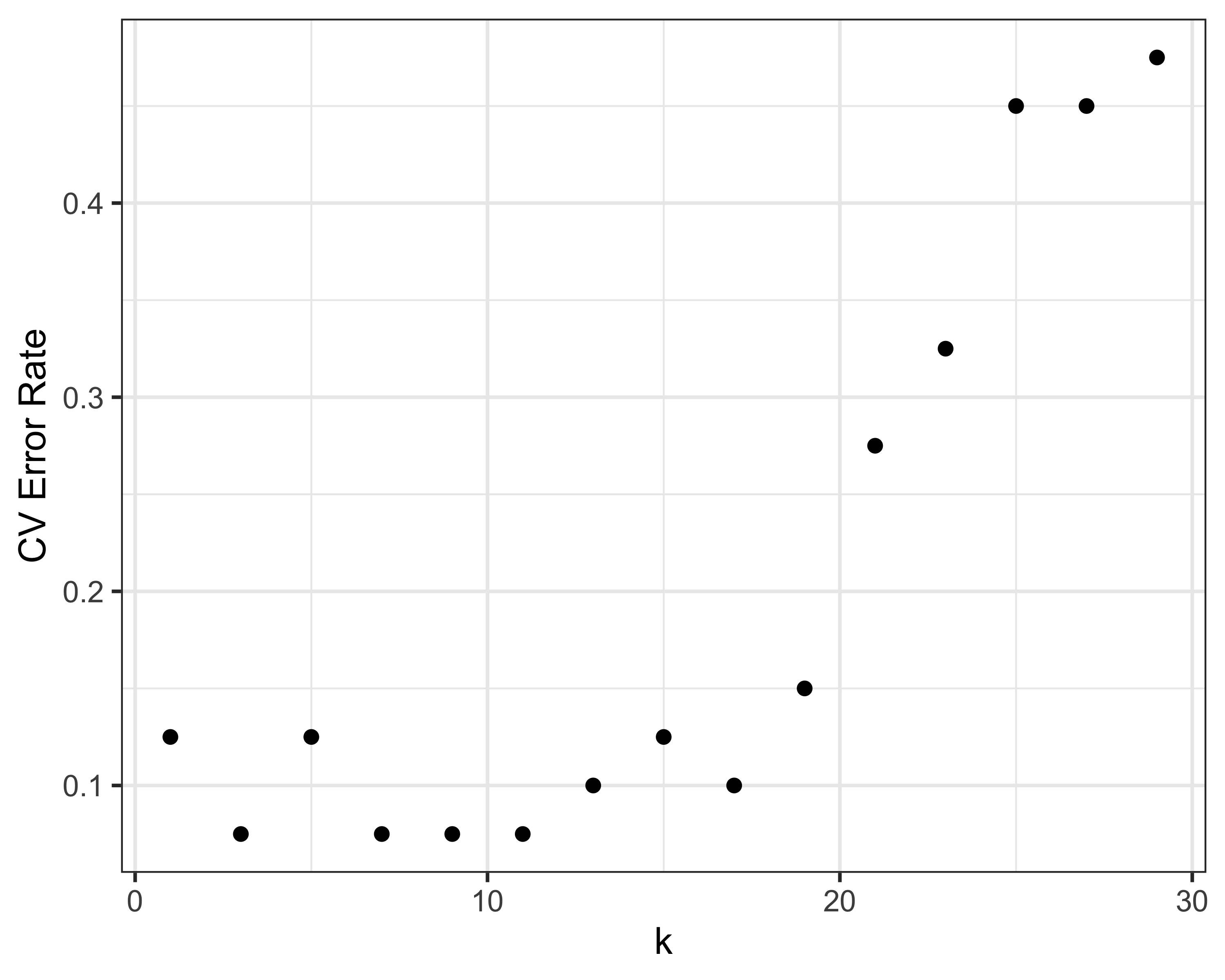
## c. Predict the tumor class for three new subjects

predict = knn(train = t(train), test = t(newpatients), cl = trainclass, k = 25)

Using , we are able to correctly predict the class for 3 out of 3 new patients.

## d. Cross validation

t = t(train)  
# Get indices  
n = 1:46  
d1 = sample(n,10)  
d2 = sample(n[!(n%in%d1)],10)  
d3 = sample(n[!(n%in%c(d1,d2))],10)  
d4 = sample(n[!(n%in%c(d1,d2,d3))],10)  
d5 = setdiff(n,c(d1,d2,d3,d4))  
# CV  
ks = ks[ks <= 30]  
es = lapply(ks, function(k){  
 kn1 = knn(train = t[c(d2,d3,d4),],test = t[d1,], cl = trainclass[c(d2,d3,d4)], k = k)  
 e1 = sum(kn1 != trainclass[d1])/10  
 kn2 = knn(train = t[c(d1,d3,d4),],test = t[d2,], cl = trainclass[c(d1,d3,d4)], k = k)  
 e2 = sum(kn2 != trainclass[d2])/10  
 kn3 = knn(train = t[c(d1,d2,d4),],test = t[d3,], cl = trainclass[c(d1,d2,d4)], k = k)  
 e3 = sum(kn3 != trainclass[d3])/10  
 kn4 = knn(train = t[c(d1,d2,d3),],test = t[d4,], cl = trainclass[c(d1,d2,d3)], k = k)  
 e4 = sum(kn4 != trainclass[d4])/10  
 mean(c(e1,e2,e3,e4))  
})  
# Plot  
p = as.data.frame(cbind(ks,unlist(es)))  
ggplot(p,aes(x=ks,y=V2)) + geom\_point() + theme\_bw() + xlab("k") + ylab("CV Error Rate")



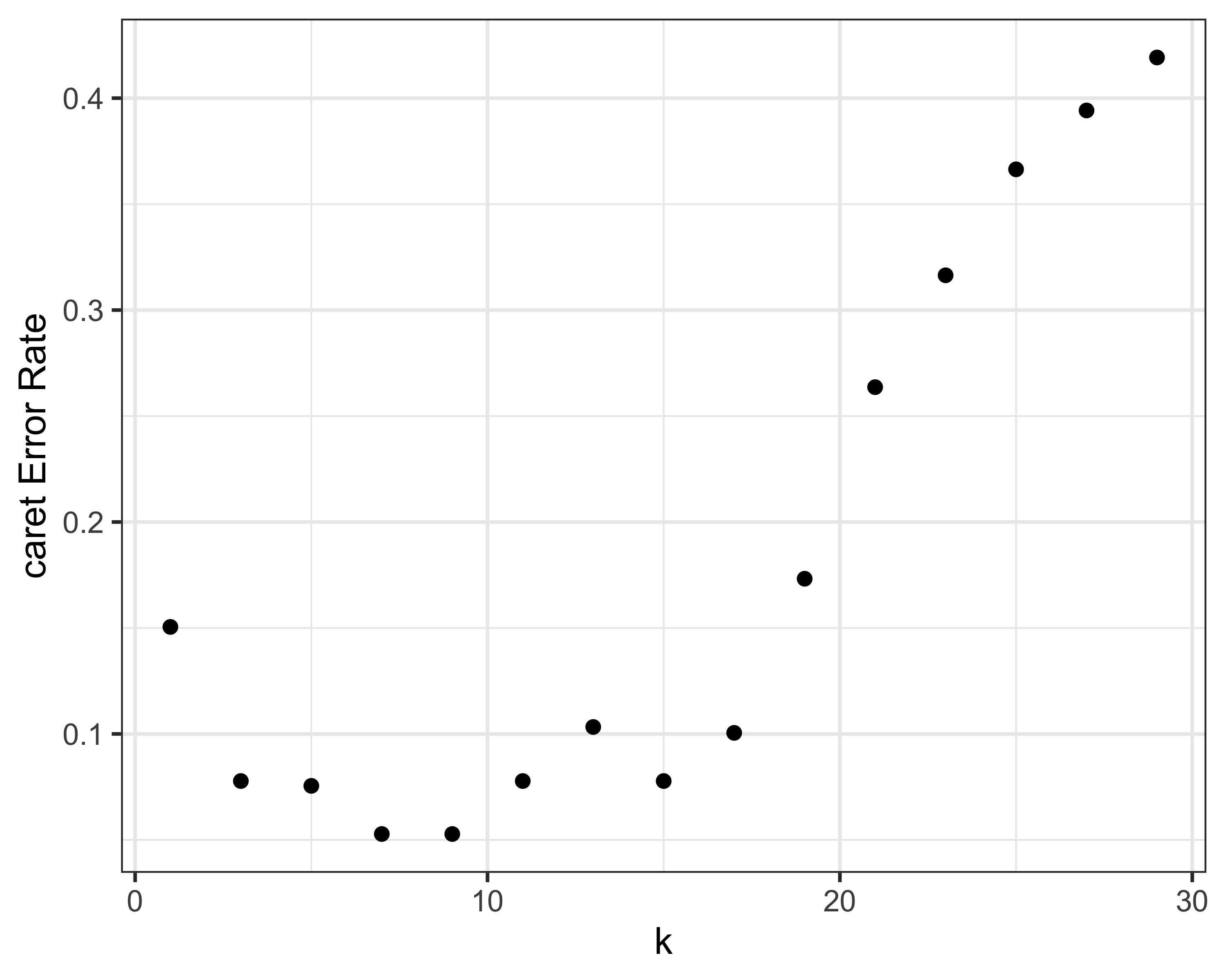
The 4-fold CV results follow a similar visual pattern overall compared to the previous section, but there are two important differences. First, is no longer matching each point to itself, so the error rate is increased. More importantly, though, is the fact that 25 no longer appears to be a good choice. In fact it’s only as good as randomly guessing between the two classes (where we’d expect an error rate of 0.5). Based on this I would choose , , or instead.

Because I’m not all that confident in my code, I decided to check it with the caret package, which has great built-in functions for validating machine learning. There is obviously a random element to how the dataset is divided into folds, so the numbers may not match up exactly, but you would hope for some agreement in terms of the choice of .

# Check with the caret package  
t = t(train)  
t = as.data.frame(t[c(d1,d2,d3,d4),])  
t$class = trainclass[c(d1,d2,d3,d4)]  
trControl <- trainControl(method = "cv",  
 number = 4)  
fit <- train(class ~ .,method = "knn",tuneGrid = expand.grid(k = ks),  
 trControl = trControl,metric = "Accuracy",data = t)  
fit$results %>% flextable(.)

| k | Accuracy | Kappa | AccuracySD | KappaSD |
| --- | --- | --- | --- | --- |
| 1 | 0.85 | 0.70 | 0.1000 | 0.199 |
| 3 | 0.92 | 0.85 | 0.0969 | 0.193 |
| 5 | 0.92 | 0.85 | 0.0510 | 0.102 |
| 7 | 0.95 | 0.90 | 0.0611 | 0.121 |
| 9 | 0.95 | 0.90 | 0.0611 | 0.121 |
| 11 | 0.92 | 0.85 | 0.0521 | 0.104 |
| 13 | 0.90 | 0.79 | 0.0912 | 0.185 |
| 15 | 0.92 | 0.85 | 0.0521 | 0.104 |
| 17 | 0.90 | 0.80 | 0.0083 | 0.014 |
| 19 | 0.83 | 0.65 | 0.0423 | 0.087 |
| 21 | 0.74 | 0.45 | 0.2507 | 0.526 |
| 23 | 0.68 | 0.34 | 0.1902 | 0.401 |
| 25 | 0.63 | 0.24 | 0.1750 | 0.364 |
| 27 | 0.61 | 0.18 | 0.1217 | 0.248 |
| 29 | 0.58 | 0.13 | 0.1330 | 0.263 |

# Plot  
p = as.data.frame(fit$results)  
p$error = 1 - p$Accuracy  
ggplot(p,aes(x=k,y=error)) + geom\_point() + theme\_bw() + xlab("k") + ylab("caret Error Rate")



The caret package would pick 9, which was also selected by my own CV code. So that’s a good sign for both my code and the choice of .

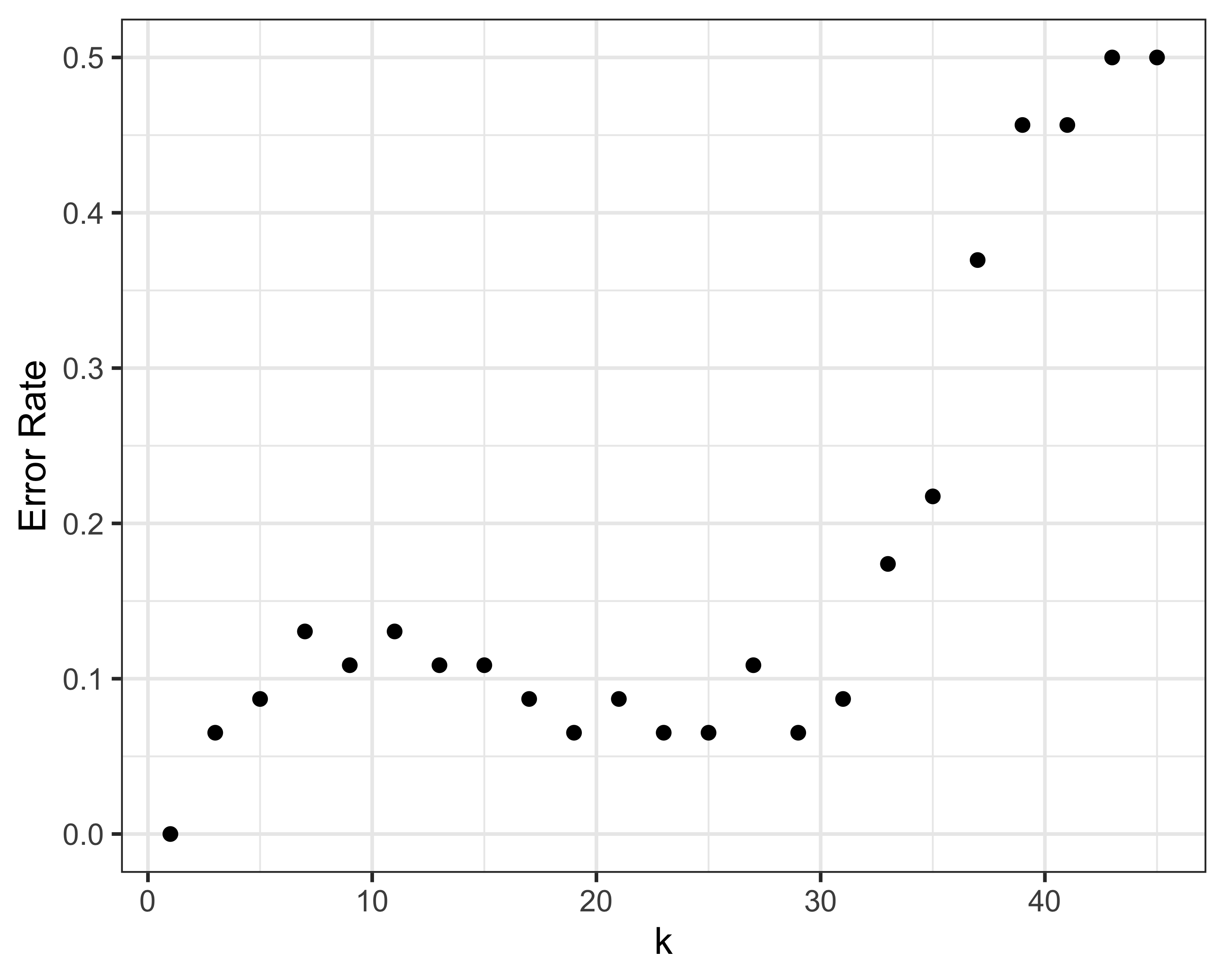
## e. Write your own code for KNN

Note that the following code is not a general-purpose KNN function, but instead assumes that the training and testing sets are the same. It also assumes no ties, so I will again only be testing with odd values of .

# This function requires that samples are in columns and genes are in rows.   
# It also assumes an odd k and does not have a tie breaking function  
knn\_tim = function(train,classes,k = 3){  
 d = as.data.frame(1 - cor(train))  
 l = lapply(d, function(s){  
 nearest = classes[order(s)[1:k]]  
 names(which.max(table(nearest)))  
 })  
 unname(unlist(l))  
}

Test multiple values for :

ks = 1:46  
ks = ks[which(ks%%2==1)]  
e = lapply(ks, function(k){  
 kn = knn\_tim(train = train,cl = trainclass,k = k)  
 sum(kn != trainclass)/length(trainclass)  
})  
p = as.data.frame(cbind(ks,unlist(e)))  
ggplot(p,aes(x=ks,y=V2)) + geom\_point() + theme\_bw() + xlab("k") + ylab("Error Rate")



This plot looks very similar to the one in part a which used Euclidean distance. Once again I think I would choose based on this training data ( would also work), but it would be good to repeat the CV process before moving forward with that choice.

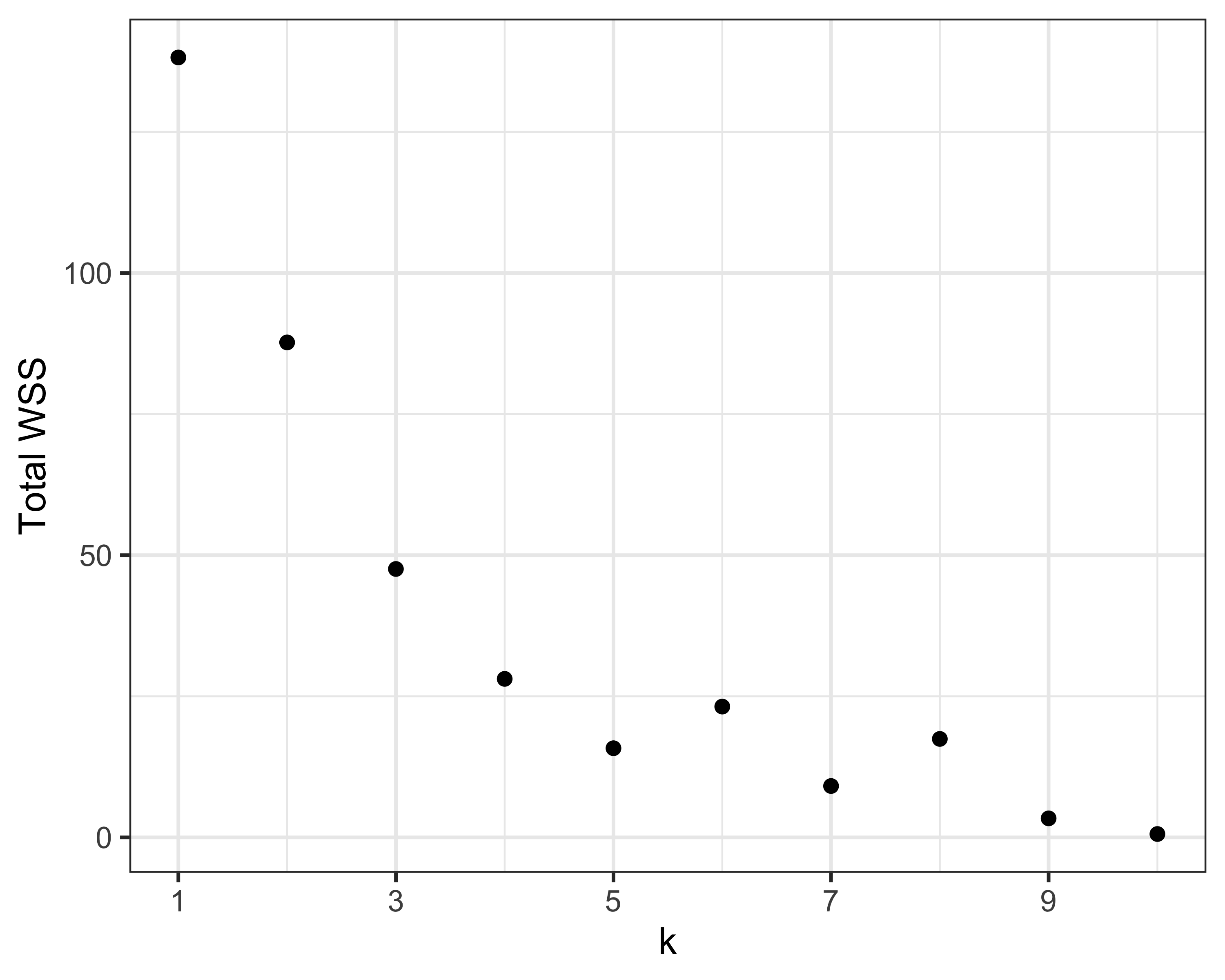
# 2. Clustering

Read in the data and standardize:

df = read.table("./data/dataHW9-cellcycle.txt")  
genes = df[,1:2]  
rownames(df) = df$V2  
df$V1 = NULL  
df$V2 = NULL  
df = t(apply(df, 1, scale))

## a. Try different values for

wss = lapply(1:(nrow(df)-1), function(k){  
 km = kmeans(df,k,iter.max = 100)  
 km$tot.withinss  
})  
# Plot  
p = as.data.frame(cbind(1:(nrow(df)-1),unlist(wss)))  
ggplot(p,aes(x=V1,y=V2)) + geom\_point() + theme\_bw() + xlab("k") + ylab("Total WSS")+  
 scale\_x\_continuous(breaks = c(1,3,5,7,9))



Based solely on this plot I think you could choose either or as the plot “elbow.” However, we already know we have three different kinds of gene in this dataset, so I’ll continue with .

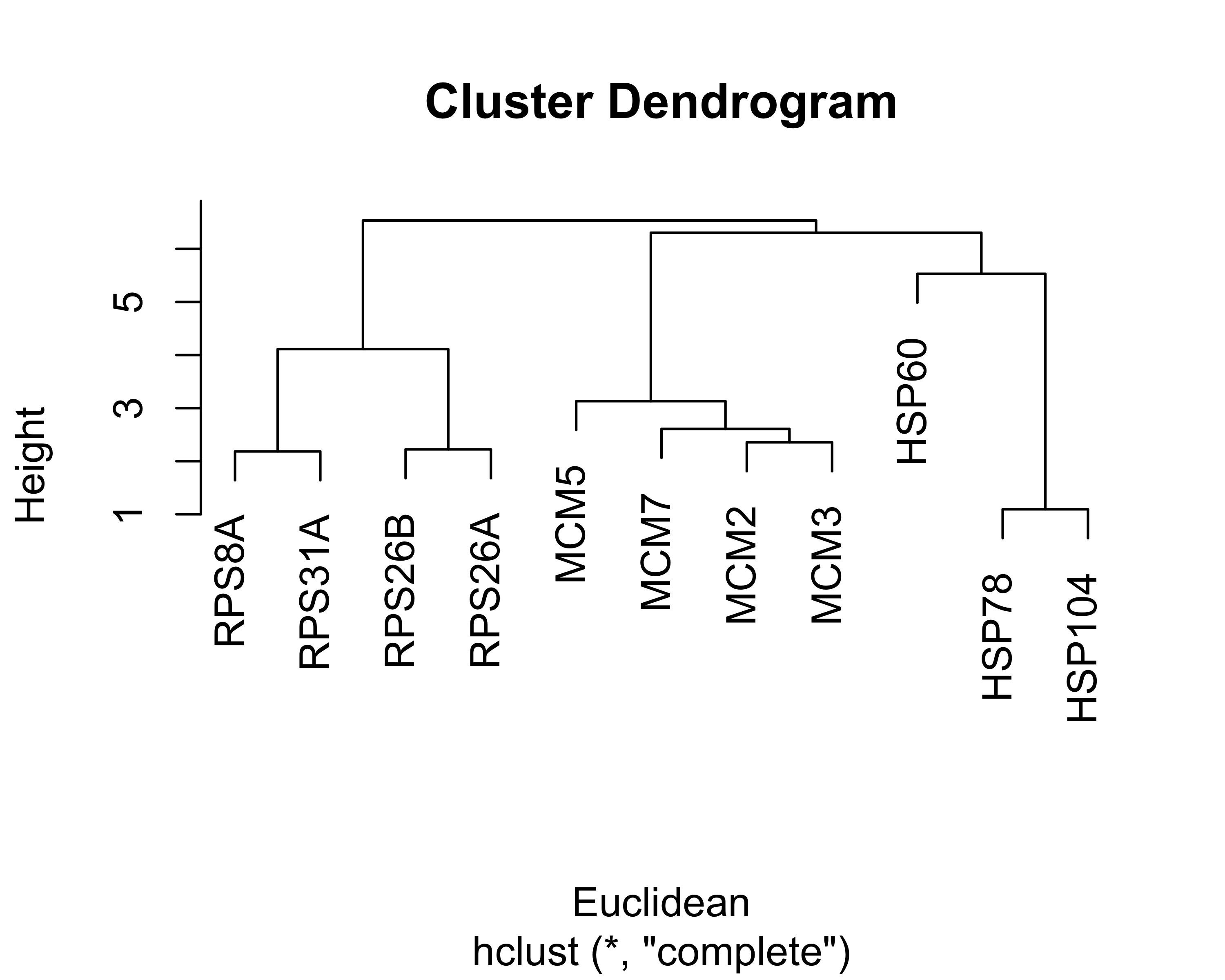
# Final clusters  
kfinal = kmeans(df,3,iter.max = 100)  
# Print  
genes$Cluster = kfinal$cluster  
genes %>% arrange(Cluster) %>% rename("Gene ID" = V1,"Gene Name" = V2) %>% flextable(.) %>%   
 set\_table\_properties(.,layout = "autofit",width = 0.5)

| Gene ID | Gene Name | Cluster |
| --- | --- | --- |
| YBL072c | RPS8A | 1 |
| YER131w | RPS26B | 1 |
| YGL189C | RPS26A | 1 |
| YGR027C | RPS31A | 1 |
| YLR259C | HSP60 | 1 |
| YBL023c | MCM2 | 2 |
| YBR202w | MCM7 | 2 |
| YEL032w | MCM3 | 2 |
| YLR274W | MCM5 | 2 |
| YDR258c | HSP78 | 3 |
| YLL026W | HSP104 | 3 |

Generally speaking the genes have clustered together by type, with genes encoding ribosomal proteins forming cluster 1, mini-chromosome maintenance complex proteins forming cluster 2, and heat shock proteins forming cluster 3. However, gene “YLR259C” clusters with the ribosomal proteins rather than the other heat shock proteins.

## b. Hierarchical clustering

d = dist(df)  
h = hclust(d)  
plot(h,xlab = "Euclidean")

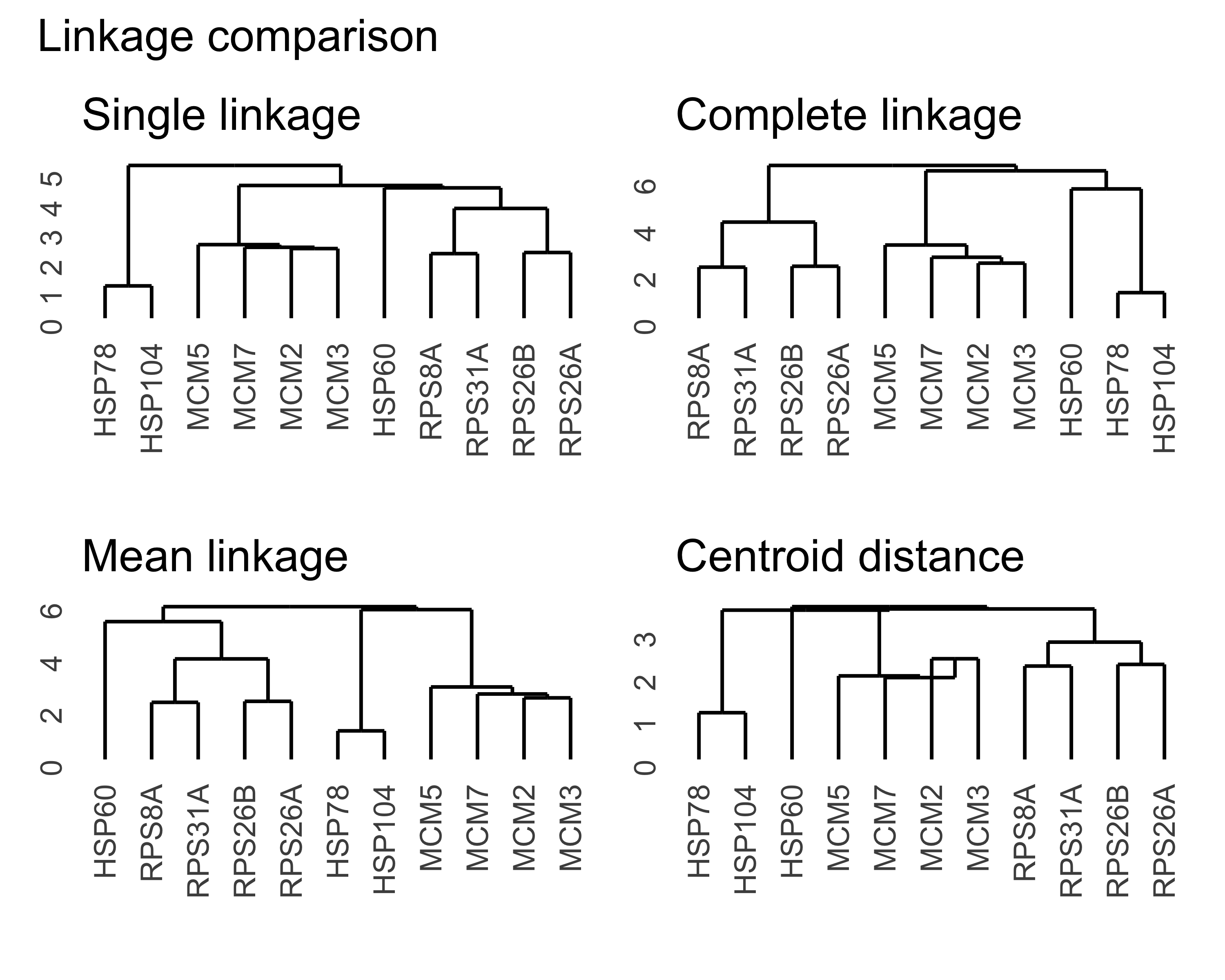


If you cut the dendrogram above at a high point (around 6), the genes are nicely clustered by type.

hclust() uses a bottom up method for creating clusters based on a dissimilarity matrix. So, each item starts in its own cluster. Then the algorithm iteratively joins the two most similar clusters and re-computes the distances between clusters until there is only one cluster left.

hclust() can use different measures of closeness when combining clusters, including complete linkage (the default), single linkage, mean linkage, and distance between centroids. Single linkage (“nearest neighbor”) combines clusters based on the minimum distance between two points in the different clusters, while complete linkage (“farthest neighbor”) is based on the maximum distance between two points. Mean linkage uses the average pairwise difference across the two clusters, and method = "centroid" uses the distance between cluster centroids.

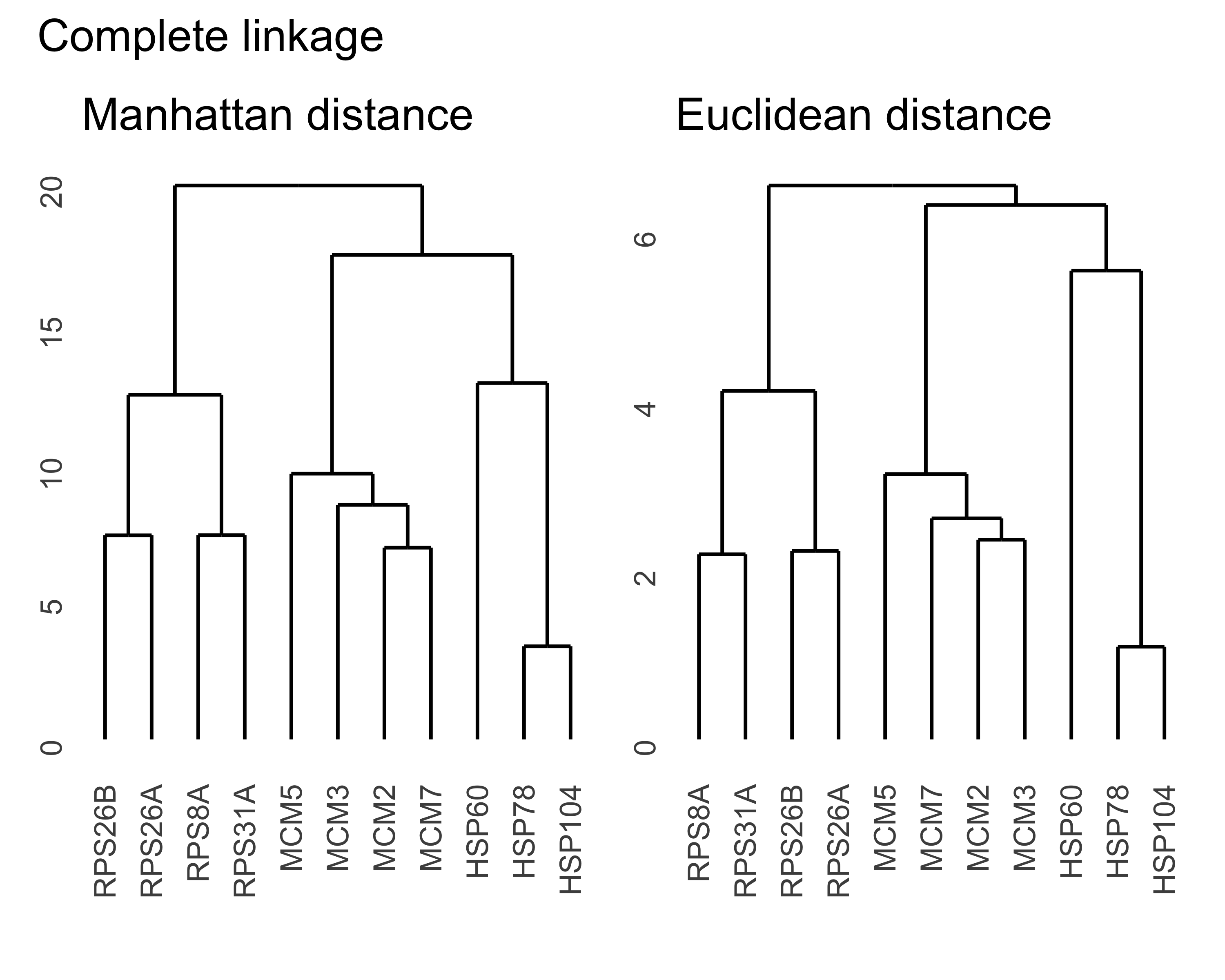
h = hclust(d,method = "single")  
single = ggdendrogram(h) + ggtitle("Single linkage")  
h = hclust(d,method = "complete")  
complete = ggdendrogram(h) + ggtitle("Complete linkage")  
h = hclust(d,method = "average")  
mean = ggdendrogram(h) + ggtitle("Mean linkage")  
h = hclust(d,method = "centroid")  
dist = ggdendrogram(h) + ggtitle("Centroid distance")  
p = (single + complete)/(mean + dist)  
print(p + plot\_annotation(title = "Linkage comparison"))



In this case the default algorithm (complete linkage) appears to be the only one that clusters the genes by type. However, the other methods generally agree with one another in terms of which genes are grouping, and these groups generally match the results of kmeans (although sometimes “HSP60” is off on its own).

### Repeat with Manhattan distance

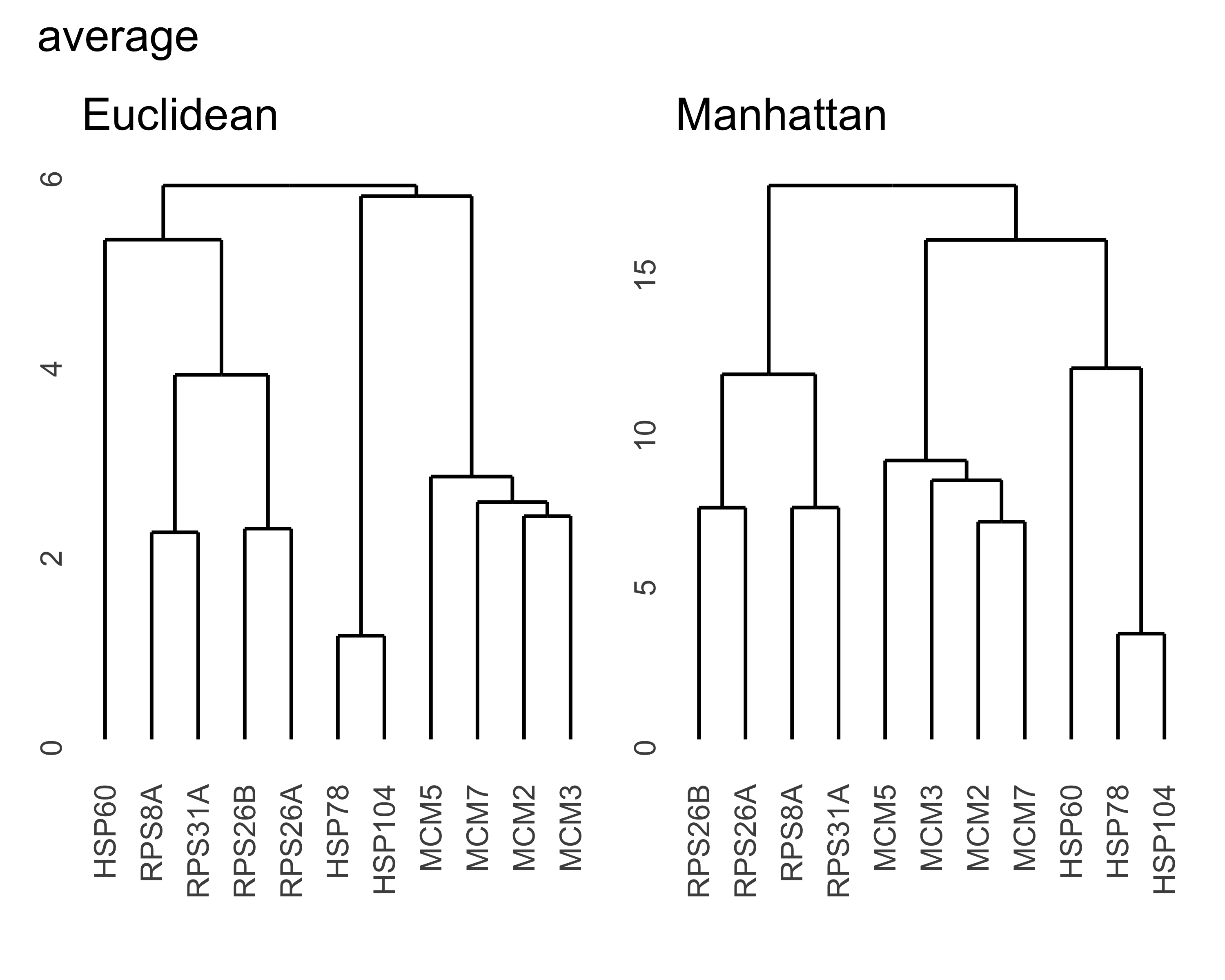
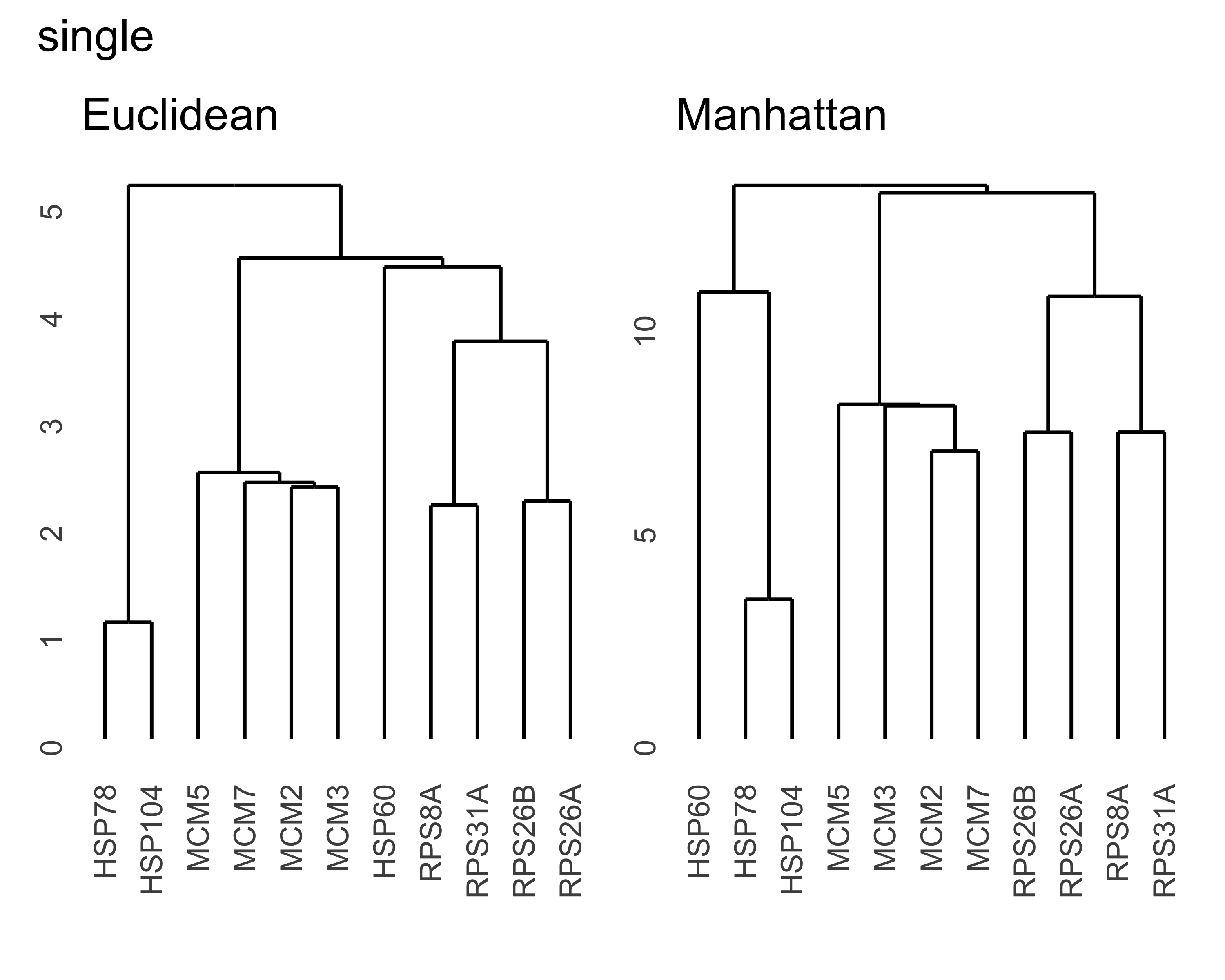
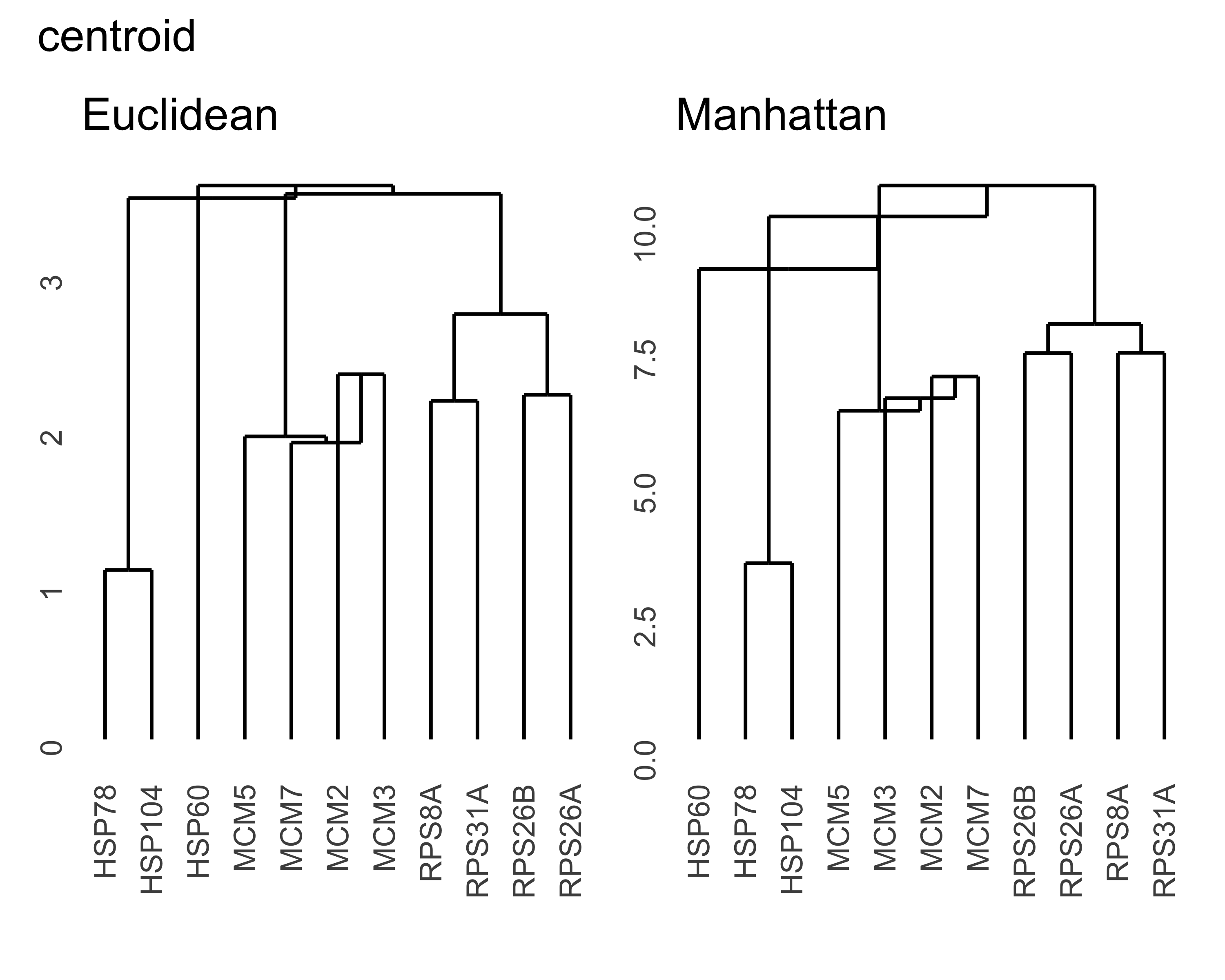
d2 = dist(df,method = "manhattan")  
h2 = hclust(d2)  
manhattan = ggdendrogram(h2) + ggtitle("Manhattan distance")  
euclidean = complete + ggtitle("Euclidean distance")  
p = manhattan + euclidean  
print(p + plot\_annotation(title = "Complete linkage"))



The dendrograms look almost exactly the same regardless of which distance metric is used, but the scale of the height is larger when using Manhattan (L1) distance.

### Compare distance metrics for different clustering methods

methods = c("centroid","single","average")  
invisible(lapply(methods, function(m){  
 h1 = hclust(d,method = m)  
 h1 = ggdendrogram(h1) + ggtitle("Euclidean")  
 h2 = hclust(d2,method = m)  
 h2 = ggdendrogram(h2) + ggtitle("Manhattan")  
 p = h1 + h2  
 print(p + plot\_annotation(title = m))  
}))



The alternative methods appear to depend more on which distance metric is used, but generally speaking the genes cluster by type regardless of the method or distance metric. The major differences have to do with how “HSP60” clusters: Sometimes it is grouped with the other heat shock protein genes, sometimes it is grouped with ribosomal proteins instead, and sometimes it is off on its own. Because the other genes don’t seem to be affected by the clustering method or distance metric, it might be worth looking into “HSP60” further to see why it appears to be different from the other genes.