Consider a remote sensing study[1] that involved detecting diseased trees in Quickbird[2] satellite imagery. Aim: find the `class` of trees (wilted/not wilted) based on the color of emitted light, as viewed from orbit. Load the training and testing datasets which can be found in the `msc` folder online.

```
D=read.csv("wilt_training.csv")
D2=read.csv("wilt_testing.csv")
str(D)
```

Find the following columns in both the training and test datasets

`class` — Factor w/ 2 levels "n","w": wilted or not

`GLCM` — num: GLCM mean texture (panchromatic band)

`Green` — num: mean in green band (520–600 nm)

`Red` — num: mean in red band (630–690 nm)

`NIR` — num: mean in near IR band (760–890 nm)

`SD` — num: standard deviation in panchromatic band

A little reconnaissance indicates a problem with these datasets:

```
table(D$class)
table(D2$class)
```

The authors say:

> we had a highly imbalanced training data set, with training data for the target class comprising only 1.7% of the total training set. Highly imbalanced training data sets have been shown to result in lower classification accuracy for the minority class

The fancy method they discussed eventually achieved an accuracy greater than 90%; you will not hit that figure. We begin with the most naive approach: KNN raw everything.

```
library(class)
predict=knn(D[,2:6],D2[,2:6],D[,1],k=1)
table(predict,D2[,1])
```

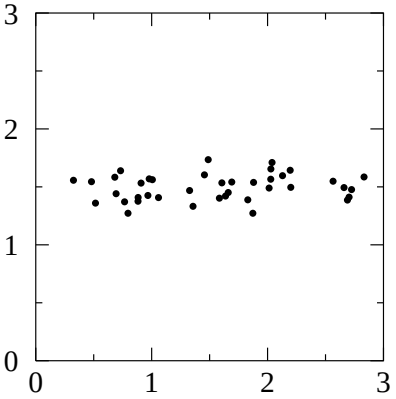
(Q1) Record (copy & paste) the table. Report the accuracy (percent on the table diagonal, i.e., those properly predicted) and the power (percent of the actually wilted predicted to be wilted) for $k = 1, 3, 5$. For all of the following models when the request is to report the 'accuracy' of a model, report all of the above (table, accuracy and power). Evidently $k = 1$ (which you would almost never use in practice) has the best accuracy: with more included neighbors we start to approach the (low) wilted fraction in the training set. The effect of more neighbors is to move true positives into false negatives; a smaller effect is to move false positives into true negatives, i.e., fewer positives generally. Note that relative tolerance for false positives vs. false negatives depends on the situation. If, for example, limited resources requires you to treat a fixed (limited) number of trees, you may be interested in maximizing the fraction of true

[1] Johnson, B., Tateishi, R., Hoan, N., 2013. International Journal of Remote Sensing, 34 (20), 6969–6982.

[2] `https://en.wikipedia.org/wiki/QuickBird`

positives among predicted positives, so that when you reach one of the limited number of trees you can treat, it is highly likely to be wilted. In this case the larger k may be desired.

It might seem that this naive approach treats each column equally, but, if one coordinate has a small range (e.g., small standard deviation), differences in that coordinate will rarely matter in determining the nearest neighbor. For example, for the below 2d dataset, the x variable will be more significant than the y variable in determining separation.



For this reason the suggestion is often to scale=standardize ($x \leftarrow \frac{x-\bar{x}}{\sigma}$) the features. Do note that to play fair the transformation of the testing set must use the same parameters $(\bar{x}, \sigma)$ that were used on the training set, i.e., $\bar{x}$ and σ are determined by the training set and applied to both training and testing datasets. R's `scale` command by default uses the data's mean and standard deviation, but can be told to use alternative `center` and `scale`. Note that it would make no sense to scale the `class` category.

```
Stdev=apply(D[,2:6], 2, sd)
Avg=colMeans(D[,2:6])
Dscaled=scale(D[,2:6])
D2scaled=scale(D2[,2:6],center=Avg,scale=Stdev)
```

Using these scaled variables is a disappointment!

```
predict=knn(Dscaled,D2scaled,D[,1],k=1)
table(predict,D2[,1])
```

(Q2) Report the accuracy for $k = 1, 3, 5$. This means that some columns have more information than others; treating them equally reduced the accuracy. How do we find the most important features? In terms of Big Data, the answer must be some automatic process, but we're not going there. Remark: You may think that the more features the better, but as discussed in the text (ISLR p. 108), more features brings on the *curse of high dimensionality*. So our plan is to find better and fewer features by hand. In the past we've discussed using `pairs` to look for relationships; `corrplot` is another option I've used. For this dataset, neither helps much; `pairs` would be helpful if could actually separately locate wilted and non-wilted datapoints. Part of the problem is 4265 non-wilted data points totally cover the few wilted data points... we need to somehow 'see through' the cloud of non-wilted cases. The details are going to get a little complicated, but hang in there. We aim to have the color and shape of the datapoints be determined by the `class`: "`n`" will be small, dim, transparent "+" (`pch`=3); "`w`" will be filled, red, opaque triangles (`pch`=24). In addition to 3 primary color dimensions, `rgb` has a fourth dimension, called α, that controls transparency. Dull gray and almost entirely transparent is `rgb(.1,.1,.1,.05)`, whereas opaque red is `rgb(1,0,0,1)`. We can put the desired color/shape values in 2-element vectors, and control which element is selected with `as.numeric(D$class)` which is 1 for `n` and 2 for `w`. (By default factors are ordered alphabetically.)

There is a lot of detail in these commands; you may want to copy & paste from the online `wilt.pdf`.

```
Color=c(rgb(.1,.1,.1,.05), rgb(1,0,0,1))
Shape=c(3,17)
pairs(D[,2:6],pch=Shape[as.numeric(D$class)],col=Color[as.numeric(D$class)])
pairs(log(D[,2:6]),pch=Shape[as.numeric(D$class)],col=Color[as.numeric(D$class)])
plot(D[,3],D[,4],pch=Shape[as.numeric(D$class)],col=Color[as.numeric(D$class)])
plot(D[,5],D[,4],pch=Shape[as.numeric(D$class)],col=Color[as.numeric(D$class)])
plot(D[,5],D[,3],pch=Shape[as.numeric(D$class)],col=Color[as.numeric(D$class)])
```

`as.numeric(D$class)` is a vector of 4339: 1 & 2; `Shape[as.numeric(D$class)]` is a vector of 4339 appropriately selected `pch`: 3 & 17.

Comment: there must be (nearly invisible) outliers far from the main clusters—the cause of all the apparent white space in a normal plot... the nearly transparent '+' are nearly invisible unless they overlap. Using log-transformation helps to focus the display on the main clumps. To my eye the cleanest separations come on `Green, Red, NIR` channels: on the Green-Red plot, the wilted points are a cluster above the non-wilted; on the NIR-Red plot, the wilted points are a cluster above the non-wilted; the NIR-Green plot doesn't look as helpful. Focus first on the Green-Red plot. (Q3) Provide a hardcopy of this plot. The majority of non-wilted lie near a line we can easily calculate by regression

```
plot(log(D[,3]),log(D[,4]),pch=Shape[as.numeric(D[,1])],col=Color[as.numeric(D[,1])])
lm(log(Red)~log(Green), data=D, subset=D$class=="n")
abline(a= -4.118, b= 1.622)
diff=log(D$Red)-(1.622*log(D$Green)-4.118)
plot(density((diff[D$class=="n"])))
lines(density((diff[D$class=="w"])),col="red")
```

(Q4) Report the R^2 of the regression. `diff` is the vertical distance between the regression line and the actual `log(Red)` value; generally the red triangles are above the line so `diff` will generally be positive for the `wilted` class. From the density plots, see the very obvious separation of wilted and non-wilted. However, a look at the *counts* in a histogram shows the problem: we've achieved a huge separation, but even in the best bins, the tail of the non-wilted predominate.

```
hist((diff[D$class=="n"]),breaks=seq(-.5,.5,.1))
hist((diff[D$class=="w"]),breaks=seq(-.5,.5,.1),col="red",add=T)
```

OK we've reduced two poor features to one good feature. The sign of a good feature is a clean separation between the classes. A 'good feature' will have the difference between the class means large compared to the scatter. If desired we could calculate means and standard deviations and put together a measure for a good feature, however the quantity we're discussing (mean difference divided by standard deviation) is what the `t.test` does. So let's perform t-tests on all the feature columns separating by `class`. The way to automate 'doing the same thing' on 'different things' is by defining a function. The 'different things' will be the argument of the function (in this case the various columns of D); the function action will be the 'same thing' (here finding the t statistic).

```
ttl=function(i){
return(t.test(log(1+D[D[,1]=="w",i]),log(1+D[D[,1]=="n",i]))$statistic)
}
sapply(2:6,ttl)
         t          t          t          t          t
  2.953412 -12.389849   2.828052  -7.335078  -4.553469
```

There is a bit of trickiness here (although the straightforward approach comes to exactly the same conclusion). I'm using `log` transformations to dampen the effect of the outliers; since some features

include zero (and log(0) =NaN), I've added one to each feature. Otherwise what's happening is fairly vanilla: `i` will determine which column we're using, and `t.test(...)$statistic` will return the desired t statistic: difference in mean divided by standard deviation. `sapply` (simple apply) will then successively load the integers between 2 and 6 into the function `ttl` and output the resulting sequence of outcomes. `Green` (which we've already used in `diff`) separates the best followed by `NIR` and `SD`. Let's put those three things together in new, dimension-reduced data.frames `d` and `d2` and see the results.

```
d=D[,c(5,6)]
d$diff=diff
d2=D2[,c(5,6)]
diff2=log(D2$Red)-(1.622*log(D2$Green)-4.118)
d2$diff=diff2
```

While equal scaling is not ideal (the `diff` separation is larger than `NIR` or `SD` separation), we'll follow the rules and scale

```
stdev=apply(d, 2, sd)
avg=colMeans(d)
dscaled=as.data.frame(scale(d))
d2scaled=as.data.frame(scale(d2,center=avg,scale=stdev))
predict=knn(dscaled,d2scaled,D[,1],k=1)
table(predict,D2[,1])
```

(Q5) Report the accuracy for $k = 1, 3, 5$. Notice that we're doing better than `Dscaled`, but not as good as the naive `D` for $k = 1$. But the accuracy and power hold up pretty well for larger k. The authors of the paper report that over-sampling a sparse training set will improve results. Let's see how we can do that. Over-sampling basically means counting the (few) wilted trees in the training set multiple times... each wilted tree in the set can stand in for 2 or 3 trees. Our `knn` output (for $k > 1$) reports the majority vote of the k nearest neighbors. Now if the $k = 5$ nearest neighbors includes just two wilted trees, the raw vote is `(n,w)`=(3,2)—which ordinarily would be $P = .6$ win for non-wilted. However if a wilted tree counts double, the vote becomes (3,4) and wilted wins. If a wilted tree gets five votes (and $k = 5$), then a raw vote `(n,w)`=(4,1) (which ordinarily would be $P = .8$ win for non-wilted) becomes (4,5) and wilted wins. So we can effectively 'over-sample' by just modifying what raw vote fraction (P) counts as a wilt: we require a super-majority ($P > .5$) for a non-wilt win.

```
predict=knn(dscaled,d2scaled,D[,1],prob=T,k=5)
predict2= (predict=="w") | (predict=="n" & attributes(predict)$prob <=.6)
table(predict2,D2[,1])
```

Notice the new `prob=T` option to output those P values. (Q6) Report the accuracy for the 2× weighed wilted trees ($P \leq 0.6$) and 5× weighed wilted trees ($P \leq 0.8$). You should find that we are now beating the $k = 5$ naive solution. Note also that (unfortunately) we've also increased the false positives. (Clearly the effect of this action is to move predicted non-wilted cases into predicted wilted—if they were wilted that improves the accuracy, if they were not wilted it increases the false positives.) Finally the accuracy we've achieved by these simple method does not match the author's accuracy using more complex methods, i.e., while we're about half-way there, there is much to learn beyond this class.