Kernel PCA

Sam Bowyer

2023-02-10

Generating The Dataset

For this portfolio we want to consider a dataset for classification that isn't linearly separable, as this will (hopefully) allow us to showcase the strengths of kernel PCA compared to regular PCA. In particular, we'll be working with two-dimensional data in the shape of concentric spirals, generated below.

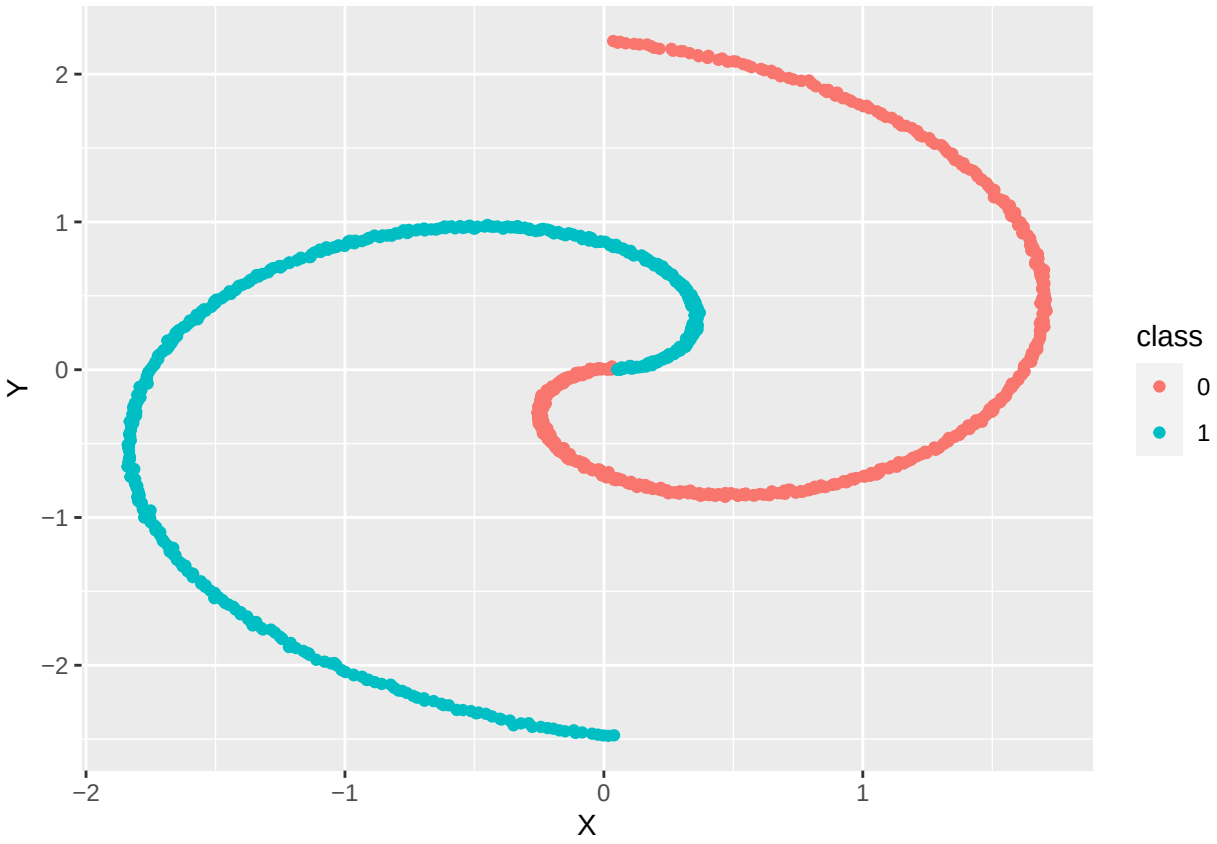
```
n = 1000  # Total number of data points

Xs = matrix(rep(0,n*2), nrow=n)

for (i in 1:n/2){
  for (class in 0:1){
    coords = c(cos(i*3*pi/(n)), sin(i*3*pi/(n))) * (class*2 -1) * ((i*(8+class))/6)
    coords = coords + rnorm(2, 0, c(2,2))   # Add some noise
    Xs[class*n/2 + i,] = coords
  }
}
Xs = scale(Xs)

# Put data into a dataframe
data = as.data.frame(cbind(Xs, c(rep(0,n/2), rep(1,n/2))))
colnames(data) = c("X", "Y", "class")
data[,"class"] = as.factor(data[,"class"])

# Plot
library(ggplot2)
ggplot(data = data, aes(X, Y, color = class)) +
  geom_point()
```

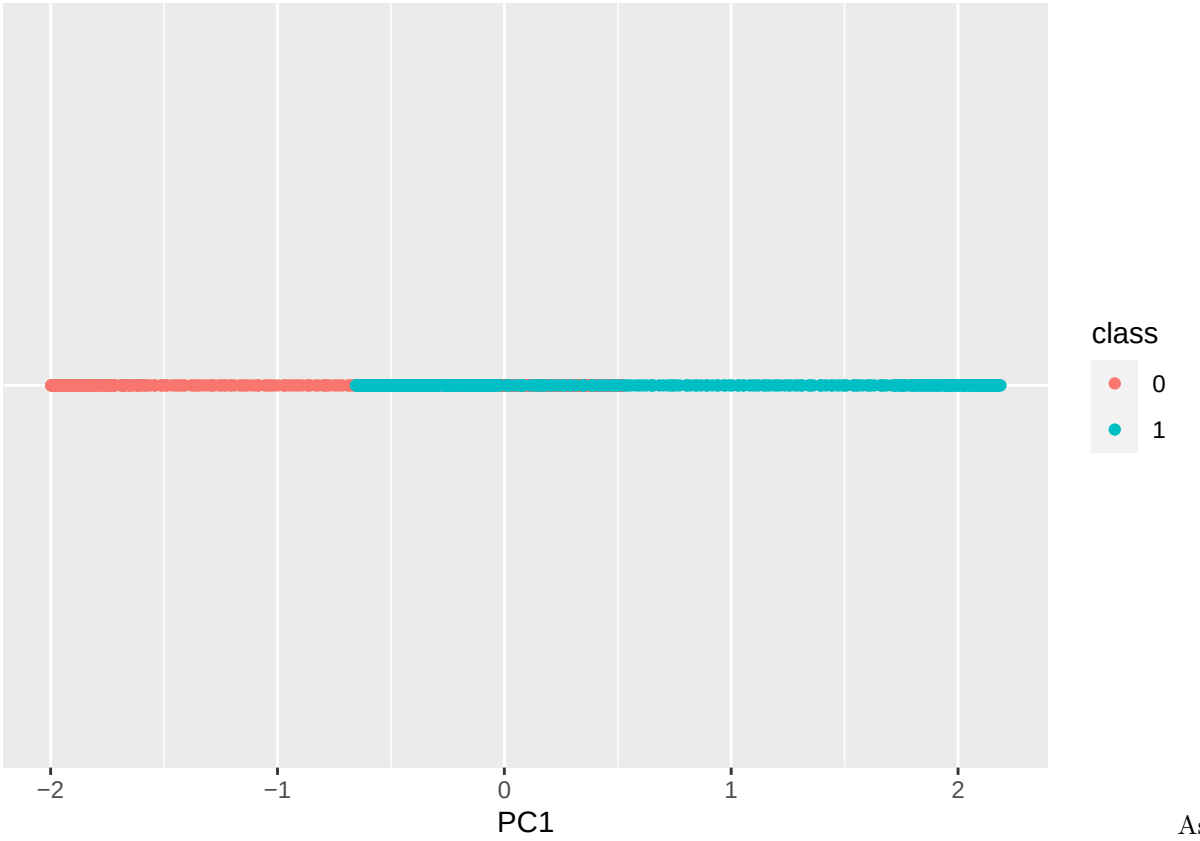


Regular PCA

If we perform regular PCA on this dataset, we can either represent the data using just one principal component or both principal components (in the latter case we might as well have not performed PCA).

```
pca = prcomp(Xs, retx=TRUE, rank=1)  # Only use first PC
reduction = data.frame(PC1=pca$x, class=data$class)

ggplot(data = reduction, aes(PC1, factor(0), color = class)) + ylab("") +
  theme(axis.text.y=element_blank(),  #remove y axis labels
        axis.ticks.y=element_blank()  #remove y axis ticks
  ) + geom_point()
```



As we might have expected, a one-dimensional reduction is unable to separate the data. Splitting the dataset into a training and test set, we can see that the classification accuracy of a logistic regression model (trained on the training set) on the test set is only 42.5%—worse than flipping a coin. (Note that we perform PCA again but only on the training set—the test set is then reduced using the same principal components found in the training set.)

```
data = data[sample(1:n, n),]  # Shuffle the data

trainSplit = 0.8
trainIdx = sample(1:n, n*0.8)
train = data[trainIdx,]
test  = data[-trainIdx,]

pca = prcomp(train[,1:2], retx=TRUE)
reduction = data.frame(PC1=pca$x[,1], class=train$class) # Only use first PC
model = glm(class ~ PC1 ,family=binomial(link='logit'), data=reduction)

prediction = predict(model,
                     newdata=data.frame(PC1 = as.matrix(test[,1:2]) %*% pca$rotation[,1]),
                     type="response")
prediction = ifelse(prediction > 0.5,1,0)
mean(prediction == test$class)  # Classification accuracy
```

```
## [1] 0.425
```

Using both principal components results in a classification accuracy of 59.5%, which is a slight improvement, however, we can do much better if we use kernel PCA.

```
reduction = data.frame(PC1=pca$x[,1], PC2=pca$x[,2], class=train$class) # Use both PCs
model = glm(class ~ PC1 + PC2, family=binomial(link='logit'), data=reduction)

prediction = predict(model,
                     newdata=data.frame(as.matrix(test[,1:2]) %*% pca$rotation),
                     type="response")
prediction = ifelse(prediction > 0.5,1,0)
mean(prediction == test$class)  # Classification accuracy
```

```
## [1] 0.595
```

Kernel PCA

To better separate the data we will now use kernel PCA (via the library `kernlab`) with the radial basis kernel $k(x, x') = \exp\left(-\frac{||\mathbf{x}-\mathbf{x}'||^2)}{2\sigma^2}\right)$ for various bandwidth values σ. One value of σ we will try is the median pairwise distance between all points in the dataset—known as the "median trick". In our dataset we find that this gives us a value of $\sigma = 1.694873$. We will also investigate the values $\sigma = 0.5$ and $\sigma = 2.5$

```
distances = rep(0, n*(n-1)/2)
count = 0
for (i in 1:n-1){
  for (j in i:n){
    count = count + 1
    distances[count] = sqrt(sum((Xs[i, 1:2] - Xs[j,1:2])^2))
  }
}
med = median(distances)
med
```

```
## [1] 1.694873
```

Before we move on to classification with the different kernel bandwidths, we can investigate the 2D PCA reduction of the entire dataset (training and test sets).

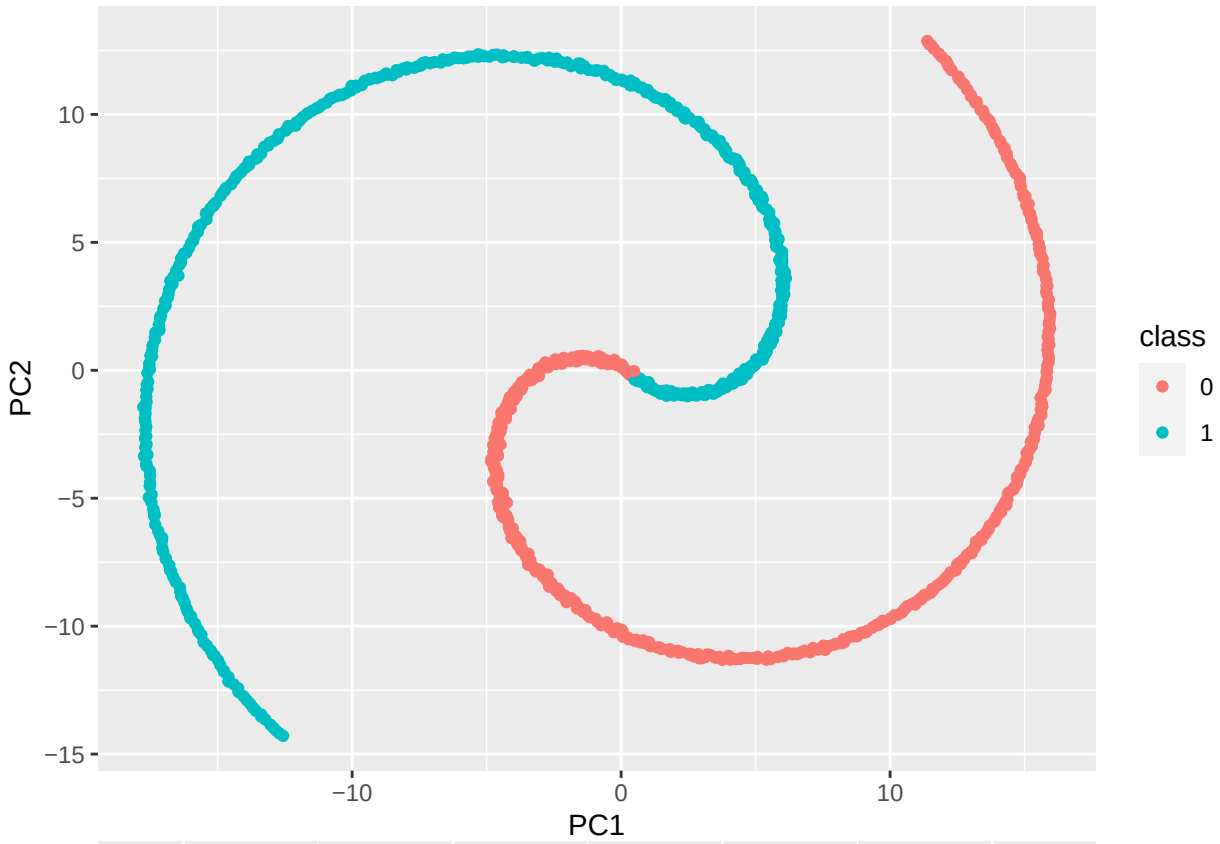
```
library(kernlab)
```

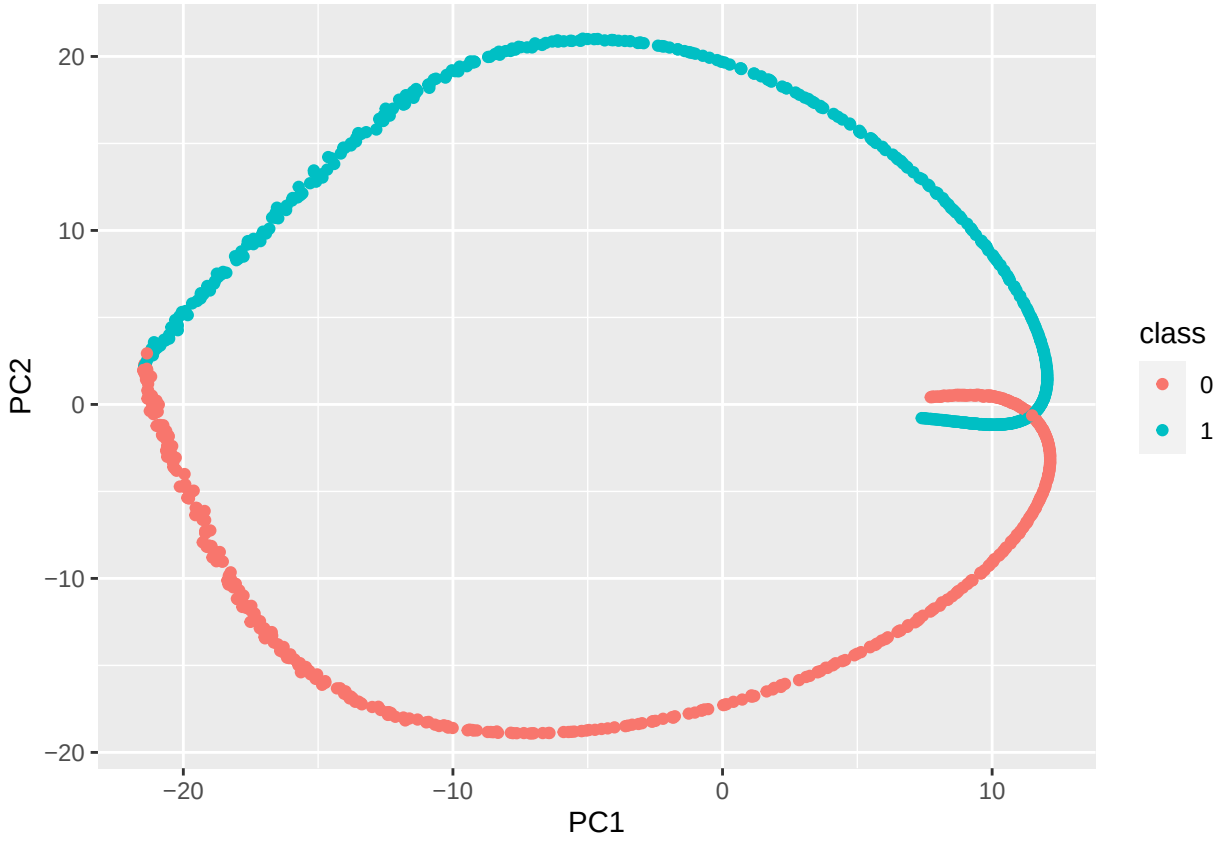
```
##
## Attaching package: 'kernlab'
```

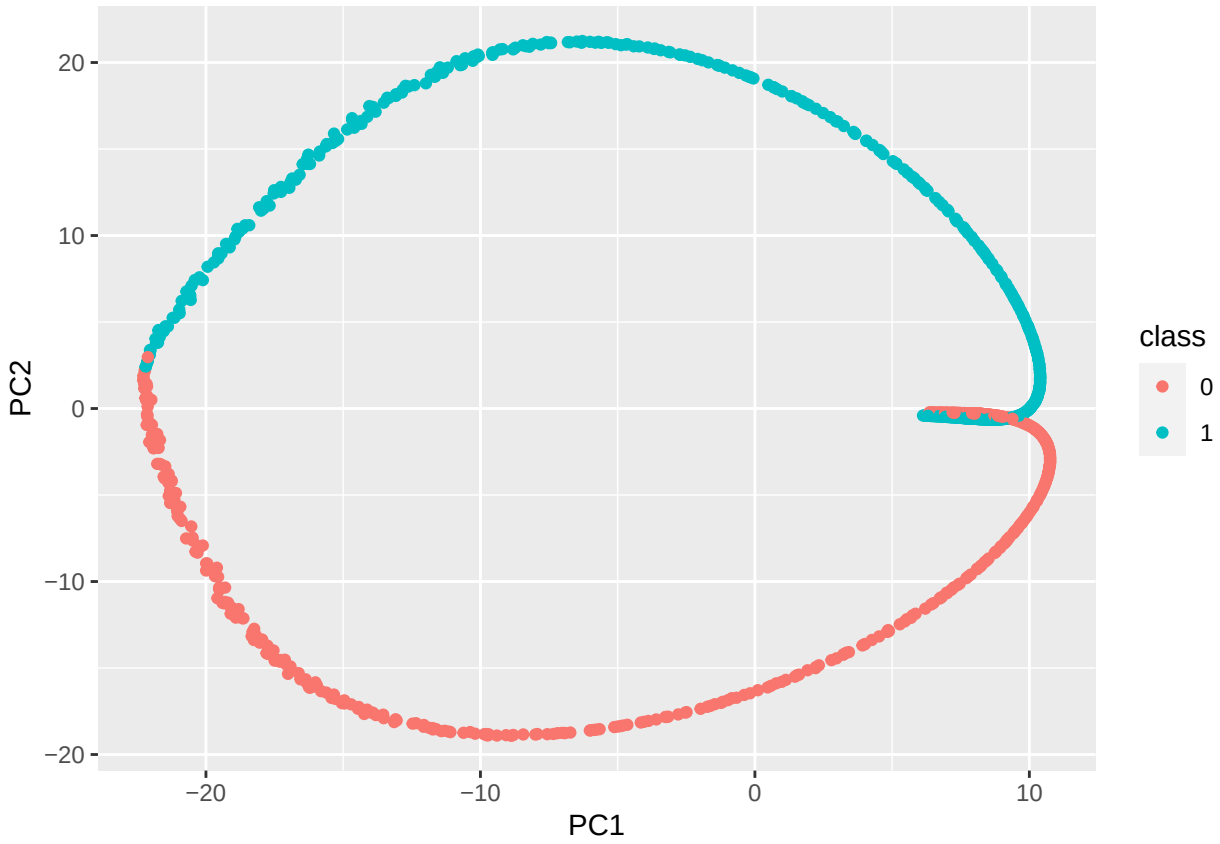
```
## The following object is masked from 'package:ggplot2':
##
##     alpha
```

```
kpca_0.5 = kpca(as.matrix(data[,1:2]), kernel = "rbfdot", kpar = list(sigma = 0.05))
kpca_med = kpca(as.matrix(data[,1:2]), kernel = "rbfdot", kpar = list(sigma = med))
kpca_2.5 = kpca(as.matrix(data[,1:2]), kernel = "rbfdot", kpar = list(sigma = 2.5))

for (k in c(kpca_0.5, kpca_med, kpca_2.5)){
  reduction = data.frame(PC1=k@rotated[,1], PC2=k@rotated[,2], class=data$class) # Use both PCs
  p = ggplot(data = reduction, aes(PC1, PC2, color = class)) +
    geom_point()
  print(p)
}
```


10
5
0
−5
−10
−15
PC2
−10
0
10
PC1
class
0
1


20
10
0
−10
−20
PC2
−20
−10
0
10
PC1
class
0
1



For $\sigma = 0.5$, we don't see much change in the dataset—in particular, it is still far from being linearly separable. In the other two cases we see similar projections, neither of which are fully linearly separable, but not nearly as bad as the original dataset (or the $\sigma = 0.5$ projection). Indeed whilst the first reduction gives us the same classification accuracy as before, performing logistic regression on the latter two reductions improves our classification accuracy up to a maximum of 82.5% when we use the median trick. (Again, note that we're finding the principle components of the training set and projecting the test set onto these.)

```
kpca_0.5 = kpca(as.matrix(train[,1:2]), kernel = "rbfdot", kpar = list(sigma = 0.05))
kpca_med = kpca(as.matrix(train[,1:2]), kernel = "rbfdot", kpar = list(sigma = med))
kpca_2.5 = kpca(as.matrix(train[,1:2]), kernel = "rbfdot", kpar = list(sigma = 2.5))

classificationAccuracy <- function(kpca_obj, numPCs){
  reduction = as.data.frame(kpca_obj@rotated[,1:numPCs])
  reduction$class = train$class
  model = glm(class ~ ., family=binomial(link='logit'), data=reduction)

  testReduction = as.data.frame(predict(kpca_obj, test[,1:2]))[,1:numPCs]
  testReduction$class = test$class

  prediction = predict(model, newdata=testReduction,  type="response")
  prediction = ifelse(prediction > 0.5,1,0)

  mean(prediction == testReduction$class)  # Classification accuracy
}
classificationAccuracy(kpca_0.5, 2)
```

```
## [1] 0.595
```

```
classificationAccuracy(kpca_med, 2)
```

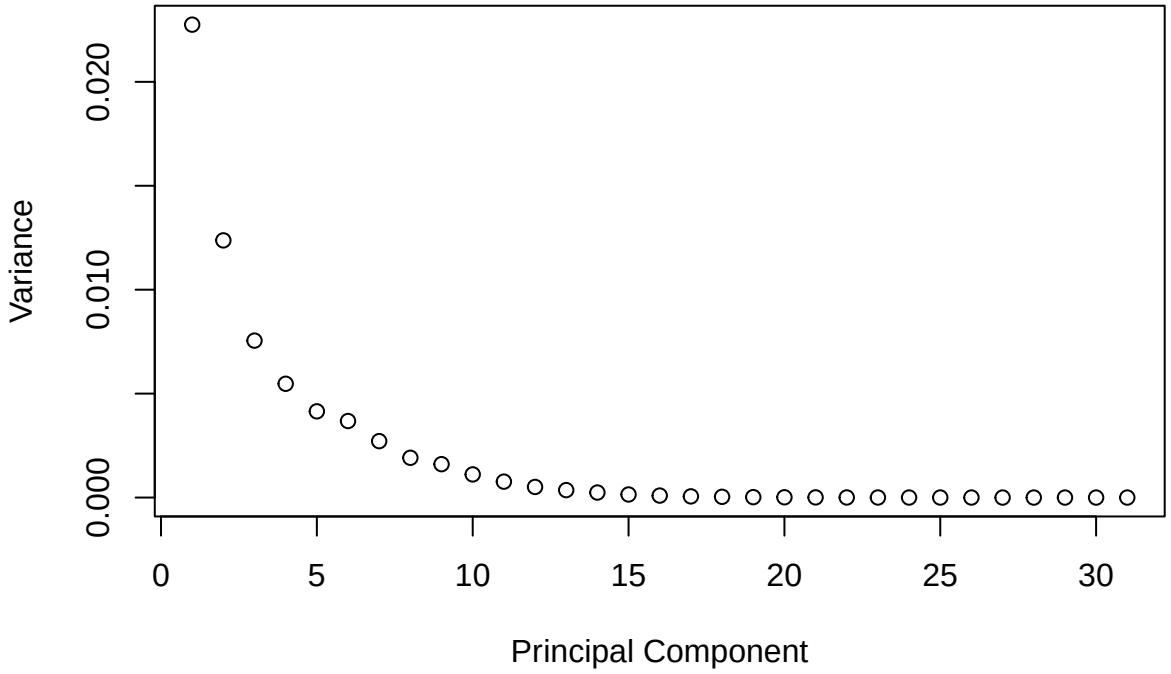
```
## [1] 0.825
```

```
classificationAccuracy(kpca_2.5, 2)
```

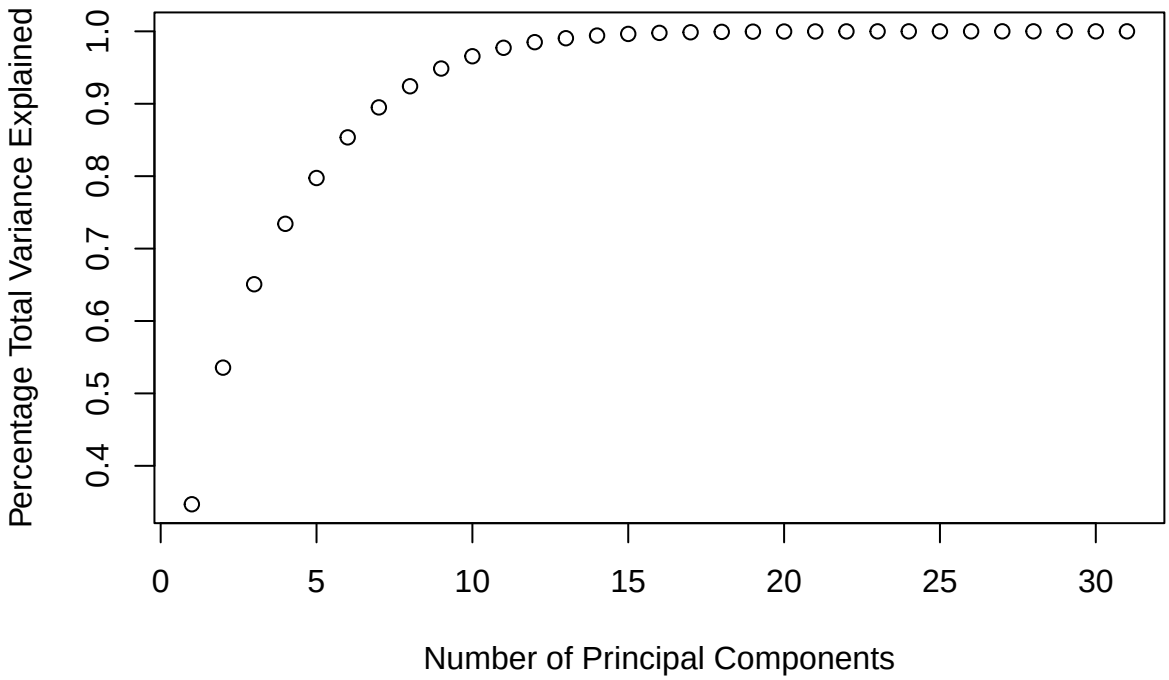
```
## [1] 0.815
```

We'll finish by finding out the classification accuracy when we use more than just the first two principal components. First we'll make a scree plot to have a look at how many principal components might be useful. (We also see that `maxPCs = 31`, i.e. the `kpca` function ignored all but 31 of the principle components as these were the ones with corresponding eigenvalues above its default value of 0.0001).

```
maxPCs = length(kpca_med@eig)
plot(1:maxPCs, kpca_med@eig^2, ylab="Variance", xlab="Principal Component")
```




```
totalVar = sum(kpca_med@eig^2)
# Or plotting cumulatively explained variance
plot(1:maxPCs, cumsum(kpca_med@eig**2 / totalVar), ylab="Percentage Total Variance Explained", xlab="Nu
```



Subsequent principal components explain less variance in the dataset than the PCs before them, as expected, however, we can see significant drop-offs in this variance after the 3rd, 6th and 10th PCs. For this reason, we'll finish this investigation by comparing the classification accuracy when using 3, 6, and 10 principle components, which explain 65.1%, 85.4%, and 96.6% of the total variance of the dataset respectively. We'll also try using just 1 principle component (which explains 34.7% of the total variance of the dataset) since this will provide a strong comparison to the one-dimensional classification performed with regular PCA earlier in the portfolio.

```
# % variance explained by first 1/3/6/10 PCs
cumsum(100*kpca_med@eig**2 / totalVar)[c(1,3,6,10)]
```

```
##   Comp.1   Comp.3   Comp.6  Comp.10
## 34.69412 65.07389 85.35937 96.56000
```

```
# Classification

# Have to deal with 1 PC case separately
reduction = data.frame(V1 = kpca_med@rotated[,1], class = train$class)
model = glm(class ~ ., family=binomial(link='logit'), data=reduction)
testReduction = data.frame(V1 = predict(kpca_med, test[,1:2])[,1], class = test$class)
prediction = predict(model, newdata=testReduction,  type="response")
prediction = ifelse(prediction > 0.5,1,0)
mean(prediction == test$class)
```

```
## [1] 0.515
```

```
# 3, 6, 10 cases
classificationAccuracy(kpca_med, 3)
```

```
## [1] 0.965
```

```
classificationAccuracy(kpca_med, 6)
```

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

```
classificationAccuracy(kpca_med, 10)
```

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

Whilst the one-dimensional case is slightly better than its non-kernel-PCA counterpart (51.5% compared to 42.5%), it's still not much better than predicting the classes uniformly at random. The impressive results when we look at using more principle components: using three obtains a 96.5% accuracy whilst using six or ten result in an accuracy of 100%.

Although we have been using a relatively small toy dataset, we have shown that kernel PCA can greatly help improve classification accuracy for simple models by allowing us to use principle components in higher dimensions than those of our original dataset. Importantly, we've also seen that (at least on our dataset) we don't need to use all that many of the (potentially-infinite) principal components to obtain a simple model that performs well—recall that using two principal components led to an 82.5% accuracy. Furthermore, it may be possible to further improve upon these results through a more thorough exploration of potential kernels: we could have tried more bandwidth parameters for the RBF kernel and we also could have tried other kernels such as the Bessel or tanh kernels.