knn_demo_auto_mpg

Nate George

January 18, 2018

```
library(data.table)
library(caret)
# a few options for KNN regression, but I went with FNN
library(FNN)
# it's a good idea to set the random seed so results are as reproducible as possible
# some functions use random initializations, so this should make them
# start in the same place every time we knit the file
set.seed(42)
```

Change the filepath to match your computer's filepath. '~' means /home/username/

```
# load data -- already replaced NAs with KNN imputation
fn <- '~/Dropbox/MSDS/MSDS680_ncg_S8W1_18/week1/auto.dt.nona.csv'

auto.dt <- fread(fn)

head(auto.dt)
```

```
##    mpg cylinders displacement horsepower weight acceleration model.year
## 1:  18         8          307        130   3504         12.0         70
## 2:  15         8          350        165   3693         11.5         70
## 3:  18         8          318        150   3436         11.0         70
## 4:  16         8          304        150   3433         12.0         70
## 5:  17         8          302        140   3449         10.5         70
## 6:  15         8          429        198   4341         10.0         70
##    origin
## 1:      1
## 2:      1
## 3:      1
## 4:      1
## 5:      1
## 6:      1
```

I almost always make separate vectors (arrays in Python) of targets and features. The targets are what we are trying to predict. The features are what we are using to predict the targets.

```
targs <- auto.dt[, mpg]
# this selects all columns except mpg
features <- auto.dt[, -'mpg', with=F]
```

Now we are creating the train and test splits. We make a training set so we can train all of our models on the train set. We can then compare how the models compare by evaluating performance on the test set. Train and tests sets should never contain any of the same data. The idea is the model has never seen the test set before, so we can check if our model is overfitting (high variance) and which model works best on unseen data. Sometimes the test data is called the 'holdout set'.

```
# create the train test split -- around 80% of data used for training
trainIdx = createDataPartition(targs, p = 0.8)$Resample1
length(trainIdx)
```

```
## [1] 321
```

```
length(targs)
```

```
## [1] 398
```

```
tr.feats <- features[trainIdx]
tr.targs <- targs[trainIdx]

# another way to get features/targets
te.feats <- auto.dt[-trainIdx, -'mpg']
te.targs <- targs[-trainIdx]
```

Now we try out the knn function. We are using regression because we are predicting on a continuous variable. We use k=3 (the default) for number of neighbors. If we give the function test data, it will return predictions under the `$pred` part of the returned value.

```
te.preds <- FNN::knn.reg(train = tr.feats, test = te.feats, y = tr.targs, k = 3)
te.preds
```

```
## Prediction:
##  [1] 17.40000 15.40000 21.66667 24.00000 13.33333 19.66667 18.50000
##  [8] 17.66667 12.66667 15.50000 11.66667 21.63333 24.66667 24.33333
## [15] 14.96667 14.83333 30.96667 15.50000 27.96667 29.33333 27.50000
## [22] 14.30000 11.66667 14.16667 33.60000 24.23333 21.30000 18.90000
## [29] 19.20000 33.70000 27.16667 28.40000 13.00000 25.40000 28.00000
## [36] 20.00000 25.80000 29.00000 18.50000 30.83333 34.63333 28.66667
## [43] 21.56667 16.83333 12.63333 15.16667 14.66667 28.06667 21.66667
## [50] 16.83333 24.16667 29.00000 23.56667 20.53333 20.73333 30.83333
## [57] 30.83333 31.60000 29.83333 23.53333 36.20000 32.56667 24.66667
## [64] 30.33333 33.03333 34.96667 25.83333 26.00000 21.56667 23.23333
## [71] 20.53333 31.73333 34.36667 30.83333 34.73333 22.33333 24.16667
```

Now we will see if we can reproduce the results to better understand the algorithm. First we get closest 3 points to the first test point. After trying this, it's not clear to me how knn.reg is getting it's predictions.

For the Euclidean distance, a proportionally similar distance with a large-magnitude number will outweigh smaller numbers, i.e. if the difference is 10% for one feature around 1000, and one feature around 1, the feature near 1000 will add around 10 to the euclidean distance, but the feature near 1 will only add something like 0.1.

```
# euclidean distance function
euc.dist <- function(x1, x2) sqrt(sum((x1 - x2) ^ 2))
dists <- c()

for (i in 1:length(tr.feats)) {
  dists <- c(dists, euc.dist(tr.feats[i], te.feats[1]))
}

# get the closest points by euclidean distance
closest <- order(dists)
top3 <- closest[1:3]
top3
```

```
## [1] 3 4 1
```

```
tr.feats[top3]
```

```
##    cylinders displacement horsepower weight acceleration model.year origin
```

```
## 1:         8          304        150   3433         12.0         70      1
## 2:         8          302        140   3449         10.5         70      1
## 3:         8          307        130   3504         12.0         70      1
```

```
te.feats[1]
```

```
##    cylinders displacement horsepower weight acceleration model.year origin
## 1:         8          318        150   3436           11         70      1
```

```
tr.targs[top3]
```

```
## [1] 16 17 18
```

```
# still don't know how it's getting the knn predictions...
# not clear which distance metric knn.reg is using or if knn.reg is normalizing data
mean(tr.targs[top3])
```

```
## [1] 17
```

```
# even weighting the mean by distance doesn't seem to
# make the prediction match the result from knn.reg
# but it is close
# possibly a slightly different weighting function is used
weighted.mean(tr.targs[top3], dists[top3])
```

```
## [1] 17.52642
```

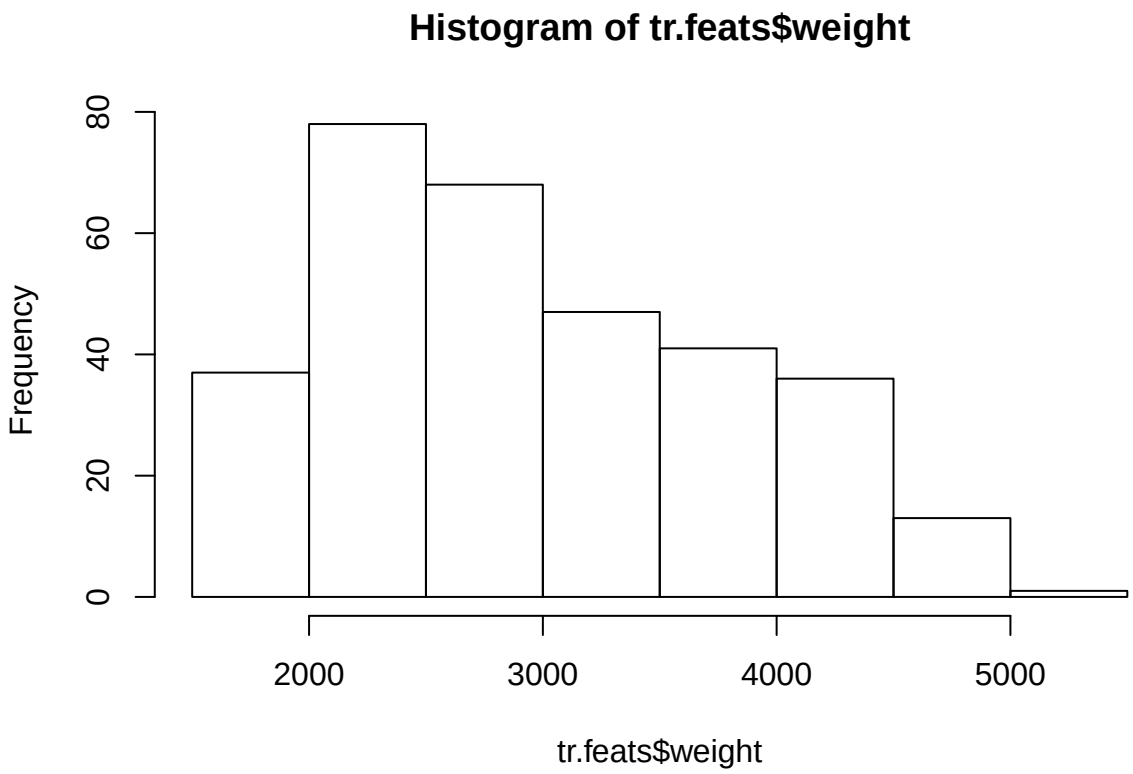
I wanted to see if normalizing the data made any difference to the knn.reg() function; it seemed to not change the performance at all. This makes me think the function is already normalizing the data.

```
# try normalizing with min/max -1 to 1
scale_range <- function(features, new.min, new.max) {
  new.feats <- copy(features)
  for (i in 1:dim(features)[2]) {
    vect <- features[, i, with=F]
    vect.max <- max(vect)
    vect.min <- min(vect)
    a <- (new.max - new.min) / (vect.max - vect.min)
    b <- new.min - (a * vect.min)
    new.vect <- a * vect + b
    new.feats[, i] <- new.vect
  }
  return(new.feats)
}

# this will get us some scoring metrics
caret::postResample(te.preds$pred, te.targs)
```

```
##      RMSE Rsquared      MAE
## 4.230841 0.718935 3.284416
```

Python advantage (maybe? I might just not know the way to do this in R) – min/max scaler or other sklearn scalers already have this built-in, also they allow you to scale on a train set and then use the parameters to scale new data (test set) without refitting anything. I think this has to be done by hand in R.

```
sc.tr.feats <- scale_range(tr.feats, -1, 1)
hist(tr.feats$weight)
```




```
hist(sc.tr.feats$weight)
```


```
te.preds.sc <- knn.reg(train = tr.feats, test = te.feats, y = tr.targs, k = 3)
te.preds.sc
```

```
## Prediction:
##  [1] 17.40000 15.40000 21.66667 24.00000 13.33333 19.66667 18.50000
##  [8] 17.66667 12.66667 15.50000 11.66667 21.63333 24.66667 24.33333
## [15] 14.96667 14.83333 30.96667 15.50000 27.96667 29.33333 27.50000
## [22] 14.30000 11.66667 14.16667 33.60000 24.23333 21.30000 18.90000
## [29] 19.20000 33.70000 27.16667 28.40000 13.00000 25.40000 28.00000
## [36] 20.00000 25.80000 29.00000 18.50000 30.83333 34.63333 28.66667
## [43] 21.56667 16.83333 12.63333 15.16667 14.66667 28.06667 21.66667
## [50] 16.83333 24.16667 29.00000 23.56667 20.53333 20.73333 30.83333
## [57] 30.83333 31.60000 29.83333 23.53333 36.20000 32.56667 24.66667
## [64] 30.33333 33.03333 34.96667 25.83333 26.00000 21.56667 23.23333
## [71] 20.53333 31.73333 34.36667 30.83333 34.73333 22.33333 24.16667
```

```
# interestingly, scaling seems to make no difference with performance
# knn.reg() is possibly already scaling the data, but there isn't an option for it
# this can be a disadvantage in R -- too much magic means we don't have as much control
# of our algorithms, and may not understand everything the algos are doing
caret::postResample(te.preds.sc$pred, te.targs)
```

```
##     RMSE Rsquared      MAE
## 4.230841 0.718935 3.284416
```

```
# have to calculate r^2 by hand in R...........
sse <- sum((te.preds.sc$pred - te.targs) ^ 2)
sst <- sum((te.targs - mean(te.targs)) ^ 2)
1 - sse/sst
```

```
## [1] 0.7188721
```

```
# if we don't give test data to knn.reg(), it will do cross-validation for us
te.preds <- knn.reg(train = tr.feats, y = tr.targs, k = 3)
te.preds
```

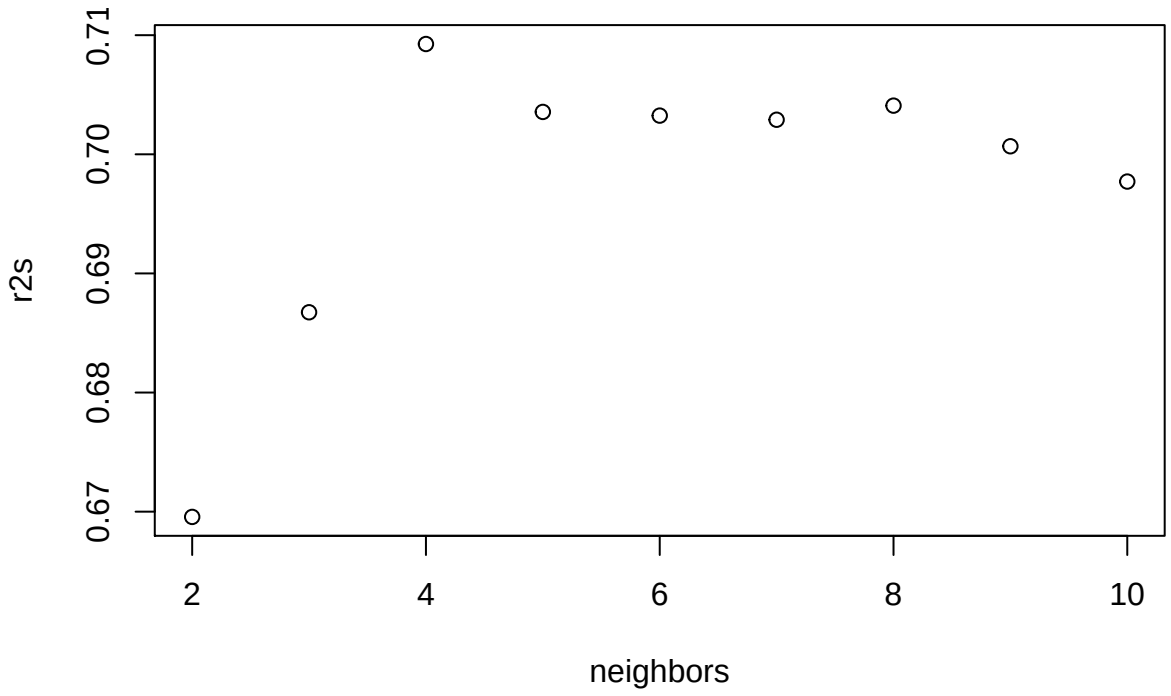
```
## PRESS =  6060.719
## R2-Predict =  0.6867384
```

```
# loop through 2-10 nearest neighbors and check cross-validation score
neighbors <- seq(2, 10)
r2s <- c()  # r-squared values (coefficient of determination)
sum.square.error <- c()  # sum of squares
for (k in neighbors) {
  preds <- knn.reg(train = tr.feats, y = tr.targs, k = k)
  r2s <- c(r2s, preds$R2Pred)
  sum.square.error <- c(sum.square.error, preds$PRESS)
}
```

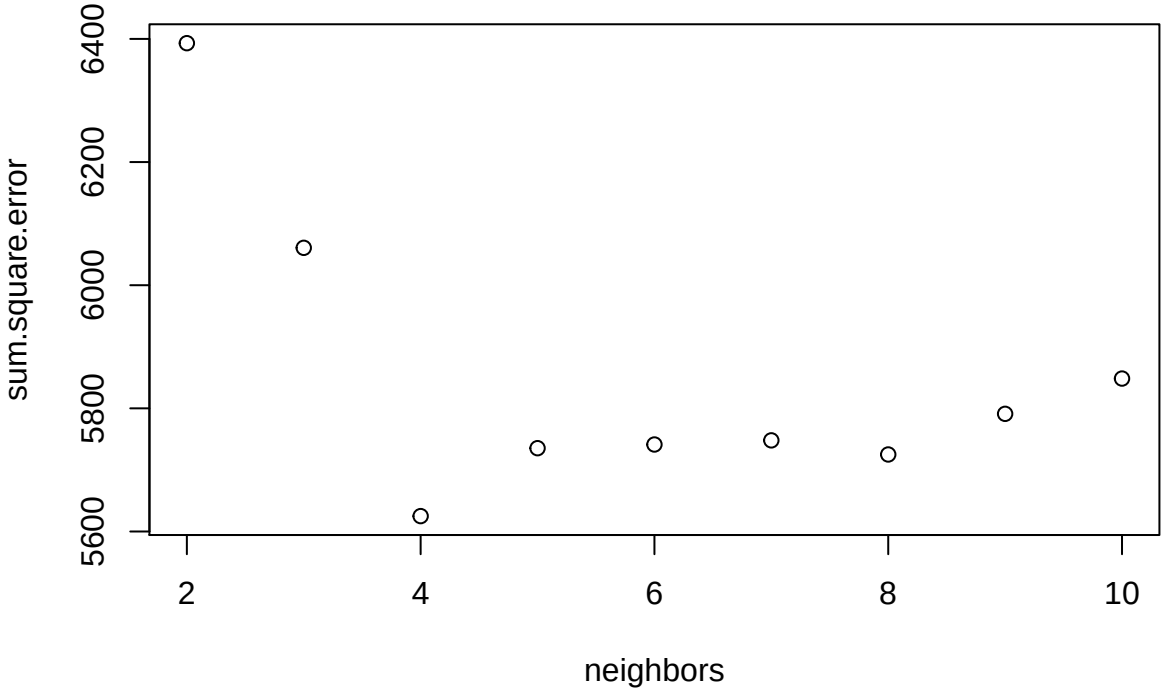
Finding the optimal number of neighbors, *k*

Typically we look for an 'elbow' in the data, where the slope changes abruptly it's pretty clearly at 5 here.

```
plot(neighbors, r2s)
```




```
plot(neighbors, sum.square.error)
```

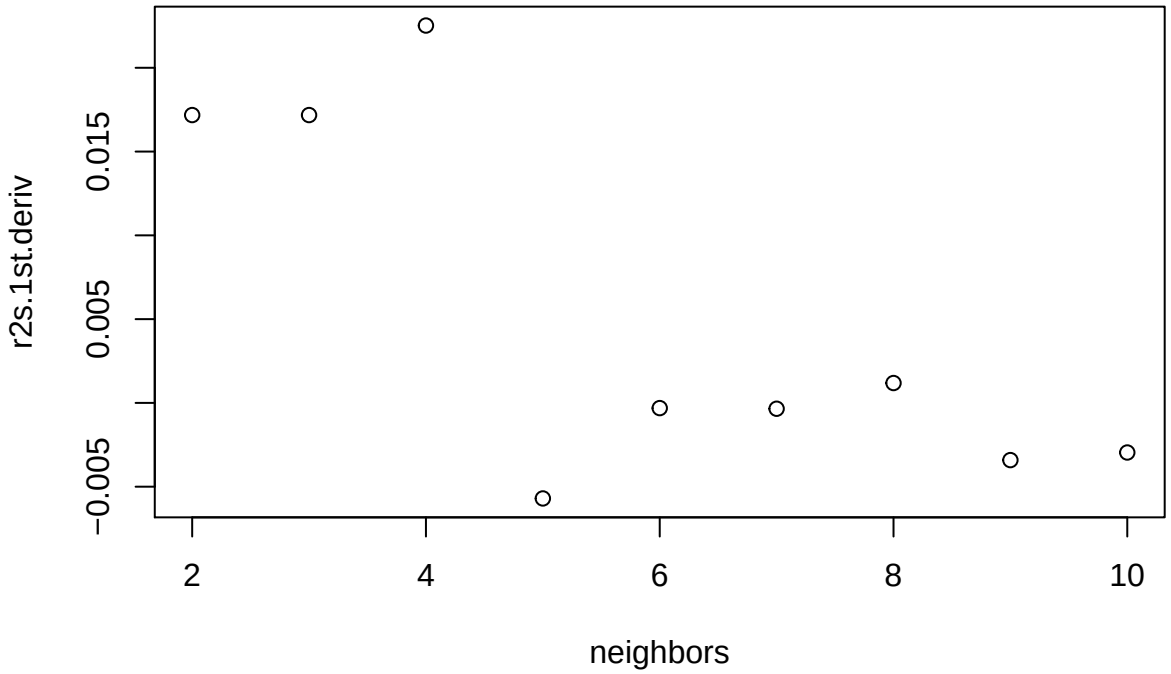



```
length(r2s)
```

```
## [1] 9
```

```
# diff gets first derivative of points
r2s.1st.deriv <- diff(r2s)
# fill in first point so we have same number as k
r2s.1st.deriv <- c(r2s.1st.deriv[1], r2s.1st.deriv)
# we can also look for when the slop gets close to or goes to 0
```

```
# and take the point just before that -- gets us k=5 again
plot(neighbors, r2s.1st.deriv)
```

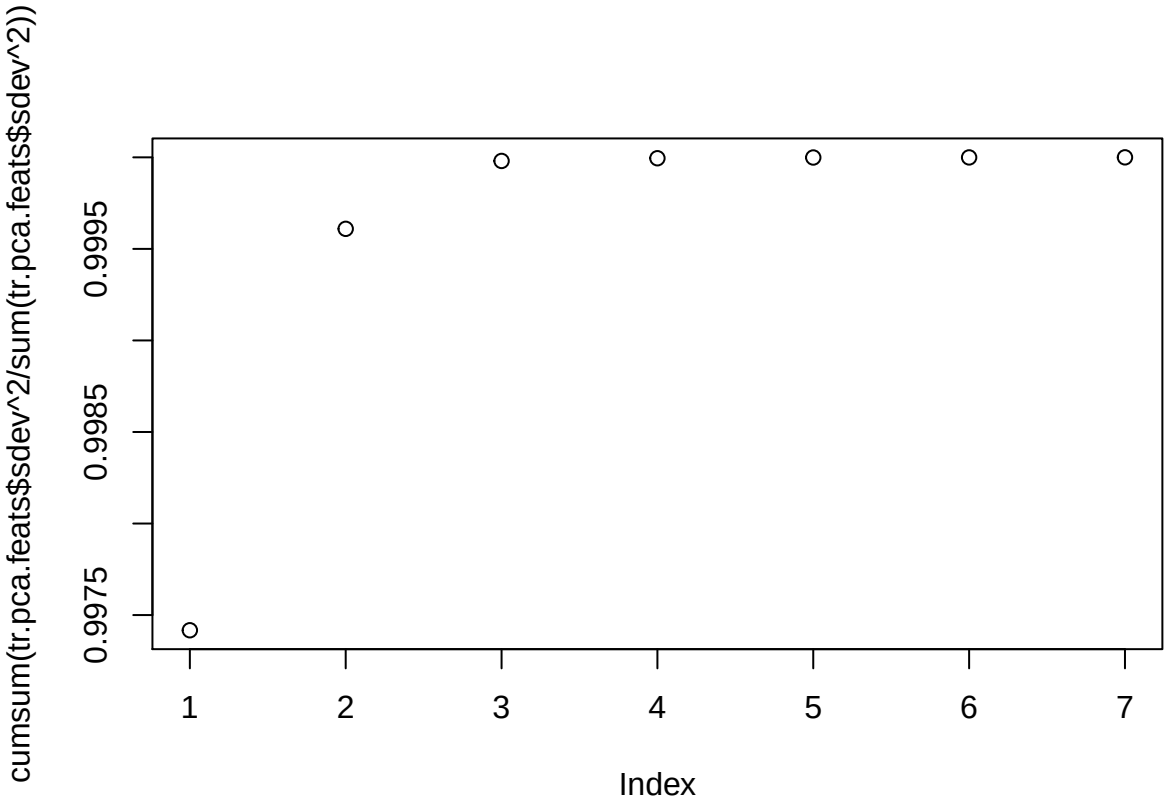


We can also use PCA to visualize high-dimensional data in 2 or 3 dimensions. PCA essentially distills data down into new features that are capturing the most variance in the data.

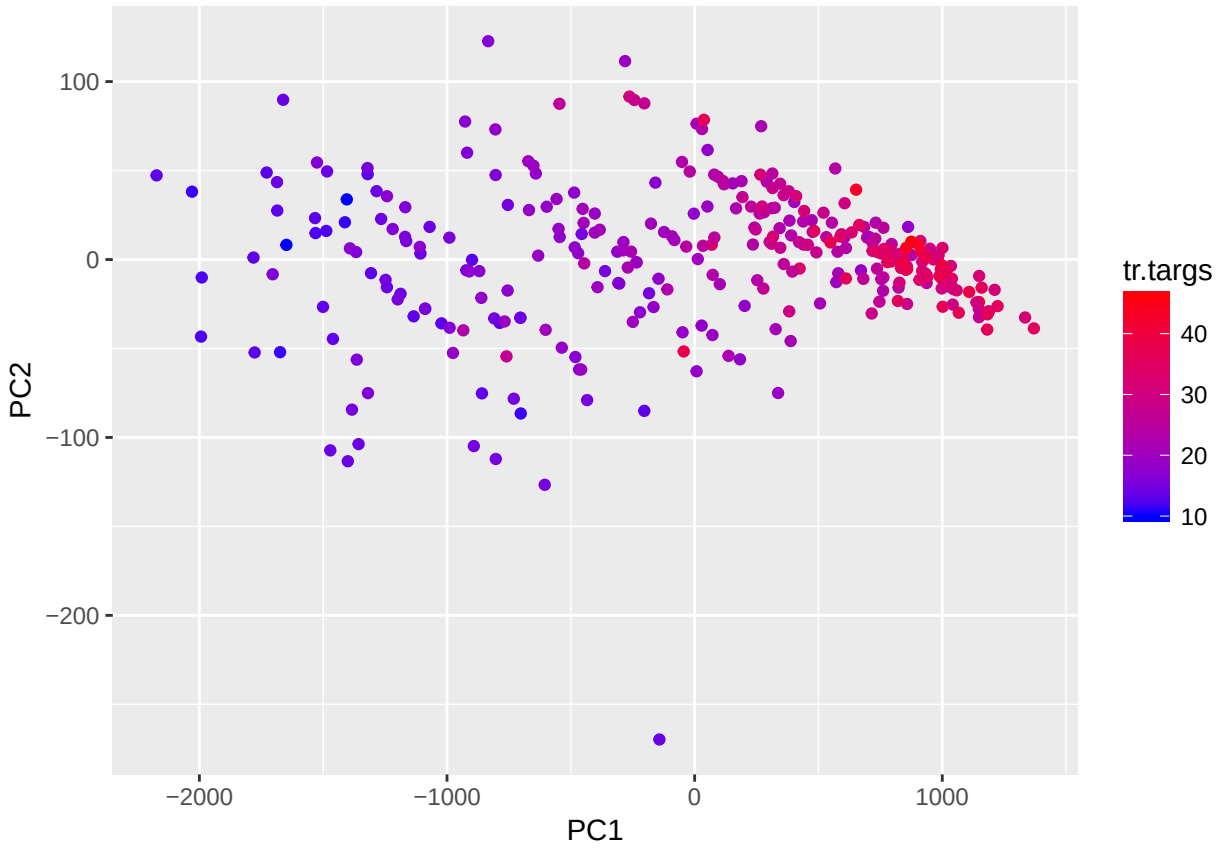
```
best.preds.cv <- knn.reg(train = tr.feats, y = tr.targs, k = 5)
print(best.preds.cv)
```

```
## PRESS =  5735.305
## R2-Predict =  0.7035582
```

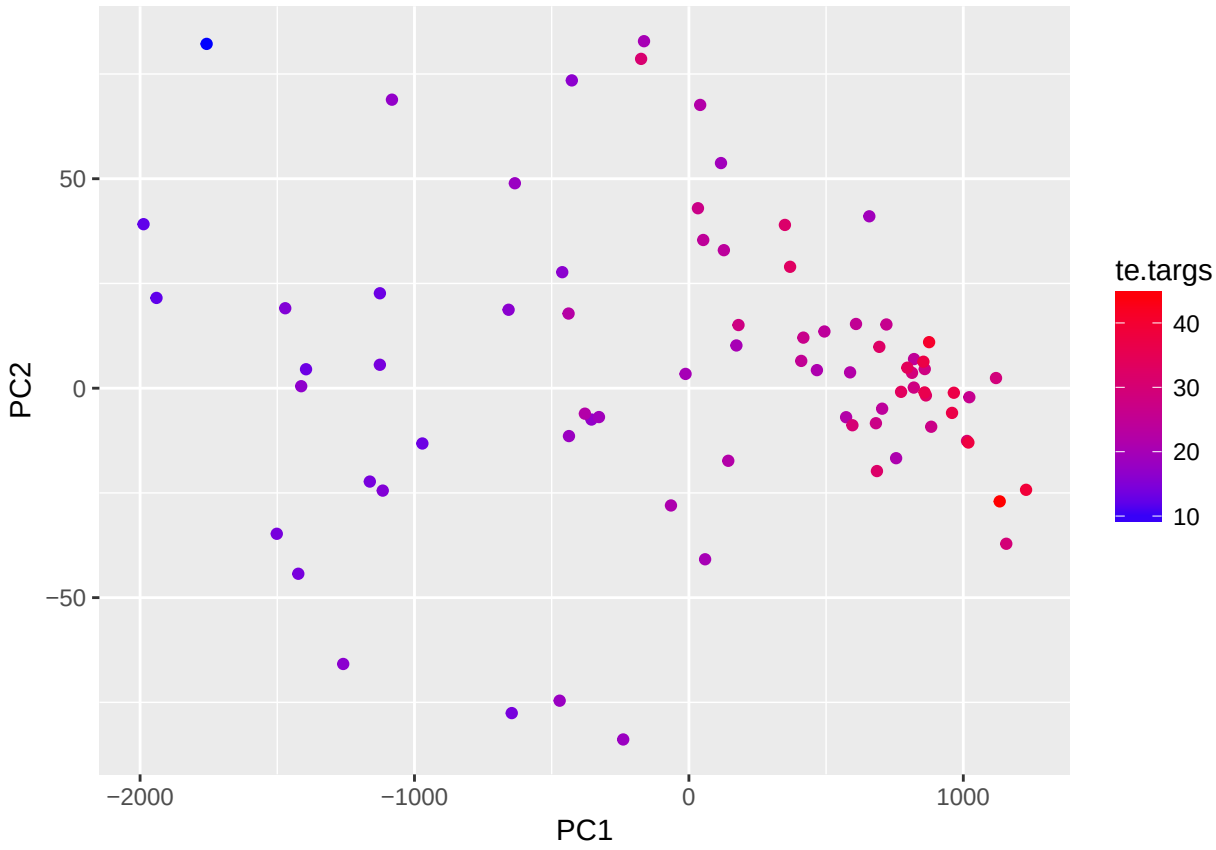
```
best.preds <- knn.reg(train = tr.feats, test = te.feats, y = tr.targs, k = 5)
te.error <- (te.targs - best.preds$pred)

tr.pca.feats <- prcomp(tr.feats)

# look at the cumulative proportion of variance of the PCA dimensions
# https://stackoverflow.com/a/24123901/4549682
# we can see in this case almost all variation in the features
# is captured by the first PCA dimension
# this is not usually like this
plot(cumsum(tr.pca.feats$sdev^2 / sum(tr.pca.feats$sdev^2)))
```



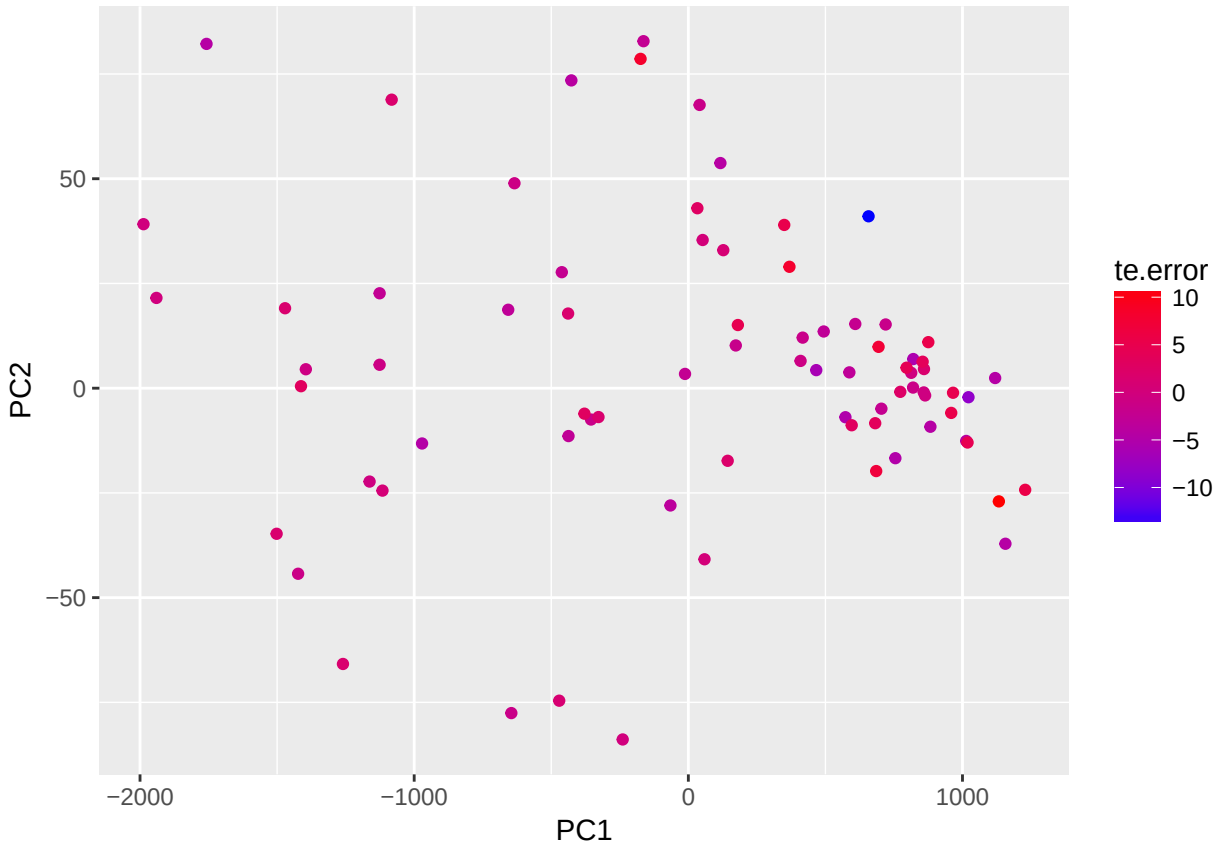

```
# http://www.sthda.com/english/wiki/ggplot2-colors-how-to-change-colors-automatically-and-manually
# this is not a great plot -- should change the labels so they are
# more informative, especially 'tr.targs'
tr.pca.dt <- as.data.table(tr.pca.feats$x)
sp <- ggplot(tr.pca.dt, aes(x=PC1, y=PC2, color=tr.targs)) +
  geom_point() +
  # Change the low and high colors
  # Sequential color scheme
  scale_color_gradient(low="blue", high="red")
sp  # shows plot
```




```
# plot the residuals of predictions in pca space
te.pca.feats <- predict(tr.pca.feats, newdata = te.feats)
te.pca.dt <- as.data.table(te.pca.feats)
sp <- ggplot(te.pca.dt, aes(x=PC1, y=PC2, color=te.targs)) +
  geom_point() +
  # Change the low and high colors
  # Sequential color scheme
  scale_color_gradient(low="blue", high="red")
sp
```




```
sp <- ggplot(te.pca.dt, aes(x=PC1, y=PC2, color=te.error)) +
  geom_point() +
  # Change the low and high colors
  # Sequential color scheme
  scale_color_gradient(low="blue", high="red")
sp
```



Next steps

Options to take it further would be to plot different variables on the x and y, or go to 3d with the features or PCA