Nonlinear and Tree-Based Models

2023-05-19

I. Introduction

We investigate techniques that allow us to model a response variable in terms of some nonlinear function of the feature space. We consider methods both for when the response is quantitative and for when the response is categorical, and we employ a different dataset in each context (one for regression, and one for classification), each retrieved from the UCI Machine Learning Repository. In the proceeding section, we consider models which extend OLS to perform a kind of nonlinear regression. Then, in the next section we consider tree-based methods, first for regression and then for classification. The analysis done here is largely informed by *An Introduction to Statistical Learning: with Applications in R*, by James et al.

II. Non-Linear Models Extending OLS

The standard linear regression model

$$\hat{y}_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip+\epsilon_i}$$

predicts the response variable Y on the basis of a linear combination of the predictors $X_1, X_2, \ldots, X_p$. However, sometimes we may desire more flexibility in order to capture nonlinear relationships between the predictors and the response variable. So, we will explore some non-linear regression models which give us this opportunity for greater complexity, and evaluate each model's performance using a train-test split and a cross-validation on the training set.

To study these methods in R, we use a dataset with 81 features extracted from 21263 superconductors along with the critical temperature of each material, the latter of which we attempt to predict. For the curious, the critical temperature is the temperature under which the material acts as a superconductor, meaning that electrical resistance vanishes and magnetic flux fields are expelled from the material, per Combescot. Attributes in the data include mean atomic mass, mean electron affinity, and mean fusion heat, among others.

First, we read in our data for regression and split our data into a training set and a test set using a 75%/25% split.

```
#load in data
super <- read.table("C:/Users/Theo/Downloads/superconduct.csv", header=TRUE, sep=",")
super <- na.omit(super)

#set a seed to ensure our data is reproducible
set.seed(123)

#create 75% training 25% testing split
regtrainIndex <- createDataPartition(super$critical_temp, p = .75, list = FALSE, times = 1)
regtrain <- super[regtrainIndex, ]
regtest <- super[-regtrainIndex, ]
```

Next, we want to use a feature selection method to narrow down the 81 predictors at our disposal. Ideally, we would perform best subsets selection using the `regsubsets()` function, which exhaustively considers every possible subset of predictors to see which is the most predictive of our response variable. Unfortunately, this is infeasible for the superconductivity dataset, as we would have to generate and evaluate

$$\sum_{k=0}^{81} \binom{81}{k} = 2^{81} \approx 2.4 \times 10^{24}$$

models, considering that there are $\binom{81}{k}$ predictor-subsets of size k.

Instead, using critical temperature as the response and the other variables as the predictors, we perform forward stepwise selection on the training set in order to identify a satisfactory model that uses only a subset of the predictors. We first fit an intercept-only model, then add one predictor at a time. Crucially, we choose which predictor to add based on which new model will provide the best improvement to our fit (i.e., yield the lowest AIC). This process continues until no added predictor will provide a significant reduction in AIC.

```
#define intercept-only model
intercept_only <- lm(critical_temp ~ 1, data=regtrain)

#define model with all predictors
all <- lm(critical_temp ~ ., data=regtrain)

#perform forward stepwise regression
forward <- step(intercept_only, direction='forward', scope=formula(all), trace=0)

#view results of forward stepwise regression
forward$anova
```

```
##                                 Step Df     Deviance Resid. Df Resid. Dev
## 1                                    NA           NA     15947   18745770
## 2      + wtd_std_ThermalConductivity -1 9699256.5244     15946    9046513
## 3           + gmean_ElectronAffinity -1  732326.2220     15945    8314187
## 4              + range_atomic_radius -1  473989.8261     15944    7840197
## 5                + std_atomic_radius -1  249322.9340     15943    7590875
## 6         + entropy_ElectronAffinity -1  305816.3025     15942    7285058
## 7       + wtd_gmean_ElectronAffinity -1  247720.0319     15941    7037338
## 8                  + wtd_std_Valence -1  169800.6486     15940    6867538
## 9        + wtd_mean_ElectronAffinity -1  159872.9782     15939    6707665
## 10        + wtd_std_ElectronAffinity -1  201912.9912     15938    6505752
## 11       + range_ThermalConductivity -1  143052.7995     15937    6362699
## 12 + wtd_entropy_ThermalConductivity -1   82413.1985     15936    6280286
## 13              + wtd_std_FusionHeat -1  107058.7317     15935    6173227
## 14               + range_atomic_mass -1   69400.1143     15934    6103827
## 15             + wtd_std_atomic_mass -1  225970.4724     15933    5877856
## 16                   + mean_Density -1   43028.1625     15932    5834828
## 17               + wtd_range_Valence -1   32012.9491     15931    5802815
## 18        + wtd_range_atomic_radius -1   58717.6760     15930    5744097
## 19                       + range_fie -1   22383.3811     15929    5721714
## 20               + wtd_gmean_Density -1   18099.9263     15928    5703614
## 21       + gmean_ThermalConductivity -1   20297.6463     15927    5683317
## 22                         + std_fie -1   19105.7152     15926    5664211
## 23    + wtd_mean_ThermalConductivity -1   19860.2326     15925    5644351
## 24   + wtd_gmean_ThermalConductivity -1   29557.8914     15924    5614793
```

```
## 25                  + wtd_range_fie -1   35864.8899     15923    5578928
## 26           + wtd_std_atomic_radius -1   22718.0481     15922    5556210
## 27   + wtd_range_ThermalConductivity -1   19162.0097     15921    5537048
## 28         + std_ThermalConductivity -1   19445.4623     15920    5517602
## 29           + wtd_range_FusionHeat -1   21959.8485     15919    5495642
## 30          + wtd_entropy_FusionHeat -1   27838.9178     15918    5467803
## 31             + wtd_entropy_Density -1   14652.2816     15917    5453151
## 32         + wtd_entropy_atomic_mass -1   19928.8364     15916    5433222
## 33             + entropy_atomic_mass -1   12176.4616     15915    5421046
## 34            + std_ElectronAffinity -1   15391.1459     15914    5405655
## 35          + range_ElectronAffinity -1  137790.7019     15913    5267864
## 36    + wtd_entropy_ElectronAffinity -1   26589.9734     15912    5241274
## 37      + wtd_range_ElectronAffinity -1   14021.4584     15911    5227253
## 38                 + wtd_entropy_fie -1   15016.3395     15910    5212236
## 39             + wtd_entropy_Valence -1   29721.0146     15909    5182515
## 40                + mean_atomic_mass -1   17971.8577     15908    5164543
## 41                   + range_Valence -1   16814.6049     15907    5147729
## 42           + wtd_gmean_atomic_mass -1    8791.0932     15906    5138938
## 43                 + std_atomic_mass -1    8232.6825     15905    5130705
## 44          + wtd_mean_atomic_radius -1    6544.4656     15904    5124161
## 45         + wtd_gmean_atomic_radius -1   45193.9254     15903    5078967
## 46             + gmean_atomic_radius -1   20244.0368     15902    5058723
## 47                + range_FusionHeat -1   10322.5792     15901    5048400
## 48                 + entropy_Valence -1    9031.2507     15900    5039369
## 49           + wtd_mean_atomic_mass -1   11936.7306     15899    5027432
## 50                  + std_FusionHeat -1    9528.7395     15898    5017903
## 51       + wtd_entropy_atomic_radius -1   10921.9288     15897    5006981
## 52              + number_of_elements -1    8537.0513     15896    4998444
## 53               + gmean_atomic_mass -1    8884.0651     15895    4989560
## 54             + wtd_mean_FusionHeat -1    4031.1980     15894    4985529
## 55           + mean_ElectronAffinity -1    3896.6394     15893    4981632
## 56                     + std_Density -1    3258.2973     15892    4978374
## 57                   + range_Density -1    8778.3969     15891    4969596
## 58                  + entropy_Density -1    3700.2536     15890    4965895
## 59                + wtd_mean_Density -1    5459.5867     15889    4960436
## 60                   + wtd_gmean_fie -1    4495.0931     15888    4955941
## 61              + entropy_FusionHeat -1    4291.9221     15887    4951649
## 62                     + entropy_fie -1    3951.5692     15886    4947697
## 63     + entropy_ThermalConductivity -1    3846.8387     15885    4943850
## 64        + mean_ThermalConductivity -1    2817.7738     15884    4941033
## 65                     + wtd_std_fie -1    1765.9760     15883    4939267
## 66                 + mean_FusionHeat -1    2079.0543     15882    4937188
## 67                + gmean_FusionHeat -1    1874.9117     15881    4935313
## 68             + wtd_gmean_FusionHeat -1    9098.8116     15880    4926214
## 69           + entropy_atomic_radius -1    2079.2877     15879    4924135
## 70                 + wtd_std_Density -1    1736.6547     15878    4922398
## 71                   + gmean_Density -1    3218.1250     15877    4919180
## 72              + mean_atomic_radius -1    2017.2976     15876    4917163
## 73           + wtd_range_atomic_mass -1    1378.0861     15875    4915784
## 74               + wtd_gmean_Valence -1    1007.0865     15874    4914777
## 75                   + gmean_Valence -1   11215.9908     15873    4903561
## 76                     + std_Valence -1     704.5863     15872    4902857
## 77                + wtd_mean_Valence -1     684.2988     15871    4902173
## 78                    + mean_Valence -1    1533.5957     15870    4900639
```

```
##          AIC
## 1  112744.63
## 2  101127.10
## 3   99782.83
## 4   98848.69
## 5   98335.30
## 6   97681.50
## 7   97131.77
## 8   96744.25
## 9   96370.60
## 10  95885.16
## 11  95532.57
## 12  95326.65
## 13  95054.45
## 14  94876.14
## 15  94276.52
## 16  94161.35
## 17  94075.61
## 18  93915.41
## 19  93855.14
## 20  93806.62
## 21  93751.76
## 22  93700.06
## 23  93646.04
## 24  93564.31
## 25  93464.11
## 26  93401.03
## 27  93347.94
## 28  93293.83
## 29  93232.23
## 30  93153.24
## 31  93112.45
## 32  93056.06
## 33  93022.28
## 34  92978.93
## 35  92569.15
## 36  92490.44
## 37  92449.72
## 38  92405.84
## 39  92316.64
## 40  92263.24
## 41  92213.24
## 42  92187.98
## 43  92164.41
## 44  92146.05
## 45  92006.77
## 46  91945.08
## 47  91914.50
## 48  91887.95
## 49  91852.13
## 50  91823.87
## 51  91791.12
## 52  91765.90
## 53  91739.53
```

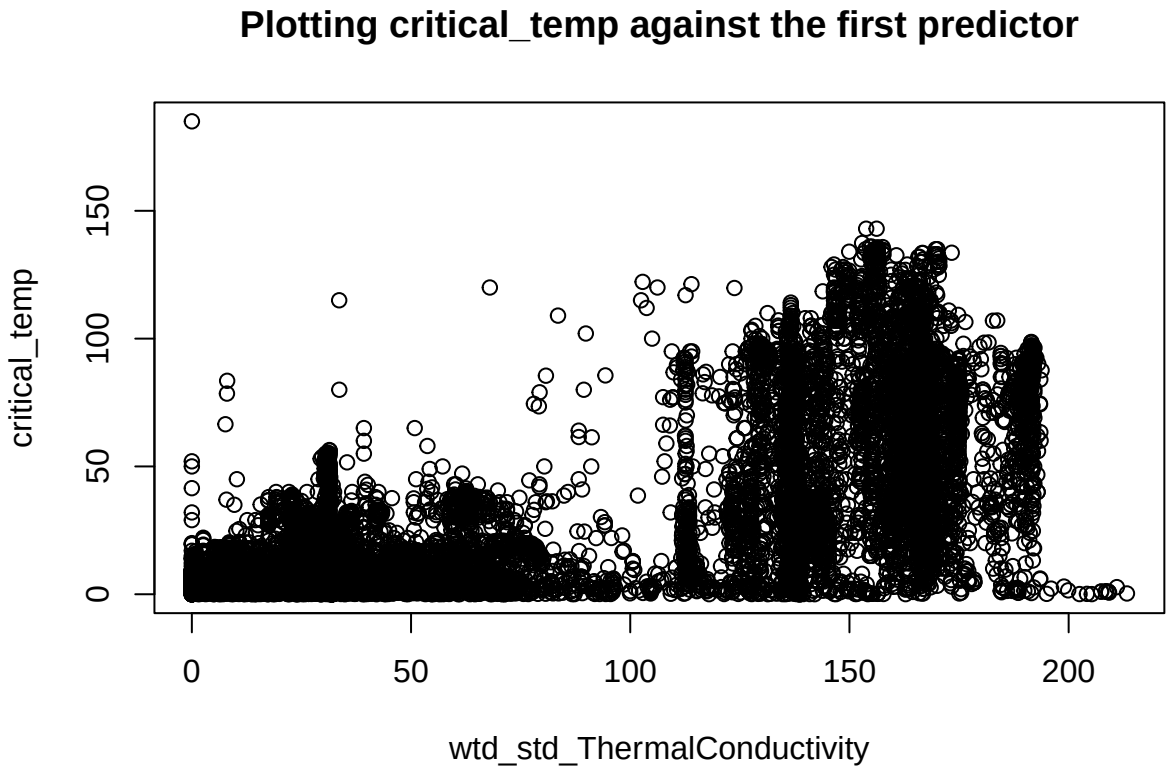
```
## 54  91728.64
## 55  91718.17
## 56  91709.74
## 57  91683.59
## 58  91673.72
## 59  91658.17
## 60  91645.71
## 61  91633.90
## 62  91623.16
## 63  91612.76
## 64  91605.67
## 65  91601.97
## 66  91597.25
## 67  91593.19
## 68  91565.77
## 69  91561.03
## 70  91557.41
## 71  91548.98
## 72  91544.44
## 73  91541.97
## 74  91540.70
## 75  91506.26
## 76  91505.97
## 77  91505.74
## 78  91502.75
```

Piecewise Polynomial

Piecewise polynomial regression allows us to split the data set into intervals based on the value of a predictor X, and then within each interval, model the data by a distinct linear combination of powers of the predictor $X, X^2, \ldots, X^d$.

In general, we perform piecewise polynomial regression on a single variable at a time. So, we look at the output from our forward stepwise selection and work with the first variable chosen, which is `wtd_std_ThermalConductivity`. We can plot this against our response, `critical_temp`:

```
plot(regtrain$wtd_std_ThermalConductivity, regtrain$critical_temp,
     xlab="wtd_std_ThermalConductivity", ylab="critical_temp",
     main="Plotting critical_temp against the first predictor")
```



It appears that the data is segmented into two distinct intervals; the cutpoint, or "knot", is right around $c = 100$. We find it exactly using the `cut()` function:

```
table(cut(regtrain$wtd_std_ThermalConductivity, 2))
```

```
##
## (-0.213,107]    (107,214]
##         7772         8176
```

We find that $c = 107$. We can now split our training dataset into two along this cutpoint and fit two different polynomial regressions, one on each segment of the data. Our model can be expressed mathematically as

$$\hat{y}_i = \begin{cases} \beta_{01} + \beta_{11} x_i + \cdots + \beta_{d_1 1} x_i^{d_1} + \epsilon_i, & \text{if } x_i \leq c\,; \\ \beta_{02} + \beta_{12} x_i + \cdots + \beta_{d_2 2} x_i^{d_2} + \epsilon_i, & \text{if } x_i > c\,, \end{cases}$$

where d_1 is the degree of the first polynomial and d_2 is the degree of the second.

Next, we want to determine the optimal degree d_k polynomial for each segment of data. Generally speaking, it is unusual to use d_k greater than 3 or 4 because for large values of d, the polynomial curve can become overly flexible and can take on some very strange shapes. So, we employ a 10-fold cross-validation on each segment to determine each optimal d_k value. Below, we print the RMSE after 10-fold cross-validation on the training set for $d_1 = 1, 2, 3, 4$ respectively and then for $d_2 = 1, 2, 3, 4$ respectively.

```
#split training set and test set along the chosen cutpoint
df_c0 <- regtrain[regtrain$wtd_std_ThermalConductivity<107,]
```

```
df_c1 <- regtrain[regtrain$wtd_std_ThermalConductivity>=107,]

test_c0 <- regtest[regtest$wtd_std_ThermalConductivity<107,]
test_c1 <- regtest[regtest$wtd_std_ThermalConductivity>=107,]

#use CV to determine optimal degree polynomial for bin df_c0
set.seed(1)
df_c0_rmse<-c()

for(i in seq(4)){
df_c0_fit <- train(as.formula(paste0("critical_temp ~ poly(wtd_std_ThermalConductivity, ", i, ")")),
                   data = df_c0,
                   method = "lm",
                   trControl = trainControl(method = "cv", number = 10),
                   na.action=na.omit)
pred_df_c0 <- predict(df_c0_fit, newdata = test_c0)
df_c0_rmse[i]<-RMSE(pred_df_c0, test_c0$critical_temp)
}

#print rmse for df_c0
df_c0_rmse
```

```
## [1] 10.50386 10.48544 10.46758 10.46486
```

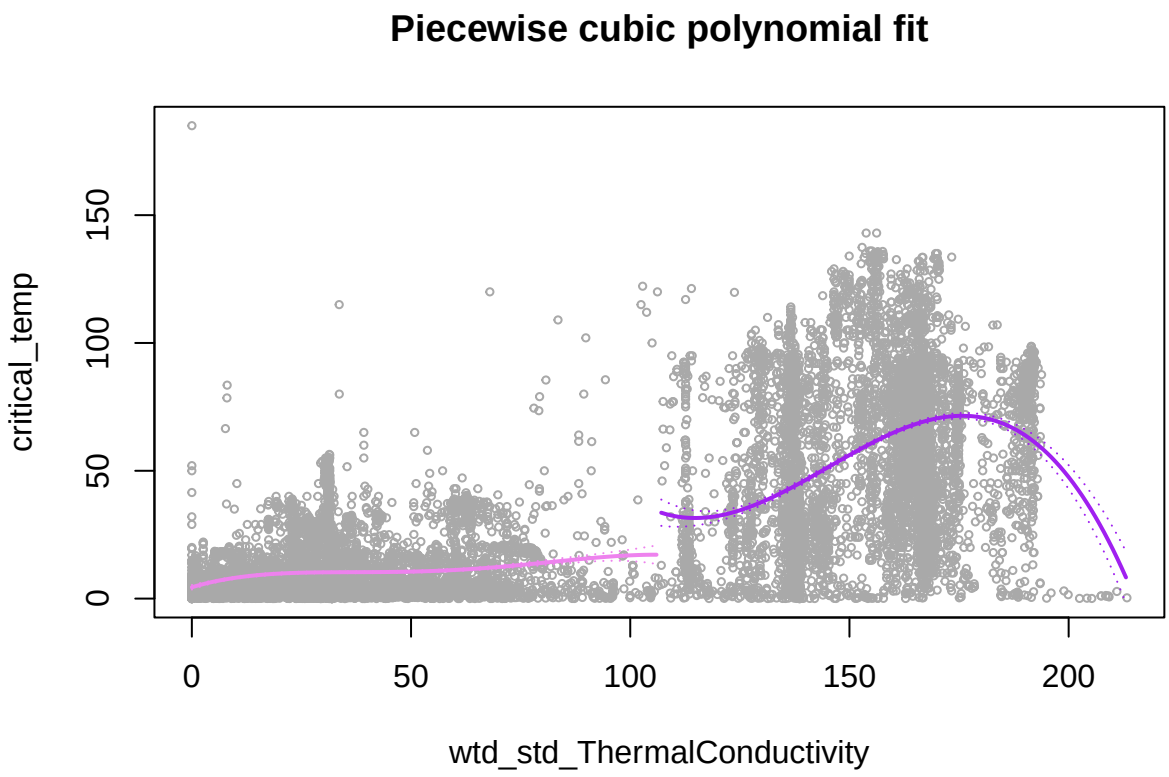
```
#use CV to determine optimal degree polynomial for bin df_c1
set.seed(1)
df_c1_rmse<-c()

for(i in seq(4)){
df_c1_fit <- train(as.formula(paste0("critical_temp ~ poly(wtd_std_ThermalConductivity, ", i, ")")),
                   data = df_c1,
                   method = "lm",
                   trControl = trainControl(method = "cv", number = 10),
                   na.action=na.omit)
pred_df_c1 <- predict(df_c1_fit, newdata = test_c1)
df_c1_rmse[i]<-RMSE(pred_df_c1, test_c1$critical_temp)
}

#print rmse for df_c0
df_c1_rmse
```

```
## [1] 30.08519 29.82405 29.67347 29.68993
```

It appears that $d_1 = 4$ is optimal on the interval with `wtd_std_ThermalConductivity<107`, while $d_2 = 3$ is optimal on the other interval. We use these d_k-values when fitting our piecewise polynomial regression models.

```
#fit one cubic polynomial regression on each subdivided training set
pwpoly0 <- lm(critical_temp~poly(wtd_std_ThermalConductivity, 4), data=df_c0)
pwpoly1 <- lm(critical_temp~poly(wtd_std_ThermalConductivity, 3), data=df_c1)

#plot the fit
```

```
c0_lims <- range(df_c0$wtd_std_ThermalConductivity)
c0.grid <- seq(from = c0_lims[1], to = c0_lims[2])
preds0 <- predict(pwpoly0, newdata = list(wtd_std_ThermalConductivity = c0.grid), se = TRUE)
se.bands0 <- cbind(preds0$fit + 2 * preds0$se.fit,
    preds0$fit - 2 * preds0$se.fit)
c1_lims <- range(df_c1$wtd_std_ThermalConductivity)
c1.grid <- seq(from = c1_lims[1], to = c1_lims[2])
preds1 <- predict(pwpoly1, newdata = list(wtd_std_ThermalConductivity = c1.grid), se = TRUE)
se.bands1 <- cbind(preds1$fit + 2 * preds1$se.fit,
    preds1$fit - 2 * preds1$se.fit)

plot(regtrain$wtd_std_ThermalConductivity, regtrain$critical_temp, cex = .5, col = "darkgrey", xlab="wt
title("Piecewise cubic polynomial fit")
lines(c0.grid, preds0$fit, lwd = 2, col = "violet")
matlines(c0.grid, se.bands0, lwd = 1, col = "violet", lty = 3)
lines(c1.grid, preds1$fit, lwd = 2, col = "purple")
matlines(c1.grid, se.bands1, lwd = 1, col = "purple", lty = 3)
```



We can now evaluate how well our fit performs on the test set.

```
#report MSE for our picewise polynomial regression
predpw0 <- predict(pwpoly0, newdata=test_c0)
predpw1 <- predict(pwpoly1, newdata=test_c1)
mse1 <- mean((predpw0-test_c0$critical_temp)^2)
mse2 <- mean((predpw1-test_c1$critical_temp)^2)
paste("RMSE of piecewise polynomial fit: ", sqrt((nrow(test_c0)*mse1+nrow(test_c1)*mse2)/nrow(regtest))
```

```
## [1] "RMSE of piecewise polynomial fit:  22.4549673846696"
```

We find that the RMSE of our piecewise polynomial fit is 22.455, so on average the value predicted by our model differs from the observed value of `critical_temp` in the test set by 22.455 K (degrees Kelvin).

Splines

Note that the piecewise polynomial model we fit above is discontinuous at our cutpoint. If we impose the constraint of continuity in derivatives up to degree $d-1$ at each knot to a piecewise polynomial regression, we now have a *spline*. A spline of degree d with K knots can be expressed as

$$\hat{y}_i = \beta_0 + \beta_1 b_1(x_i) + \beta_2 b_2(x_i) + \cdots + \beta_{K+d} b_{K+d}(x_i) + \epsilon_i$$

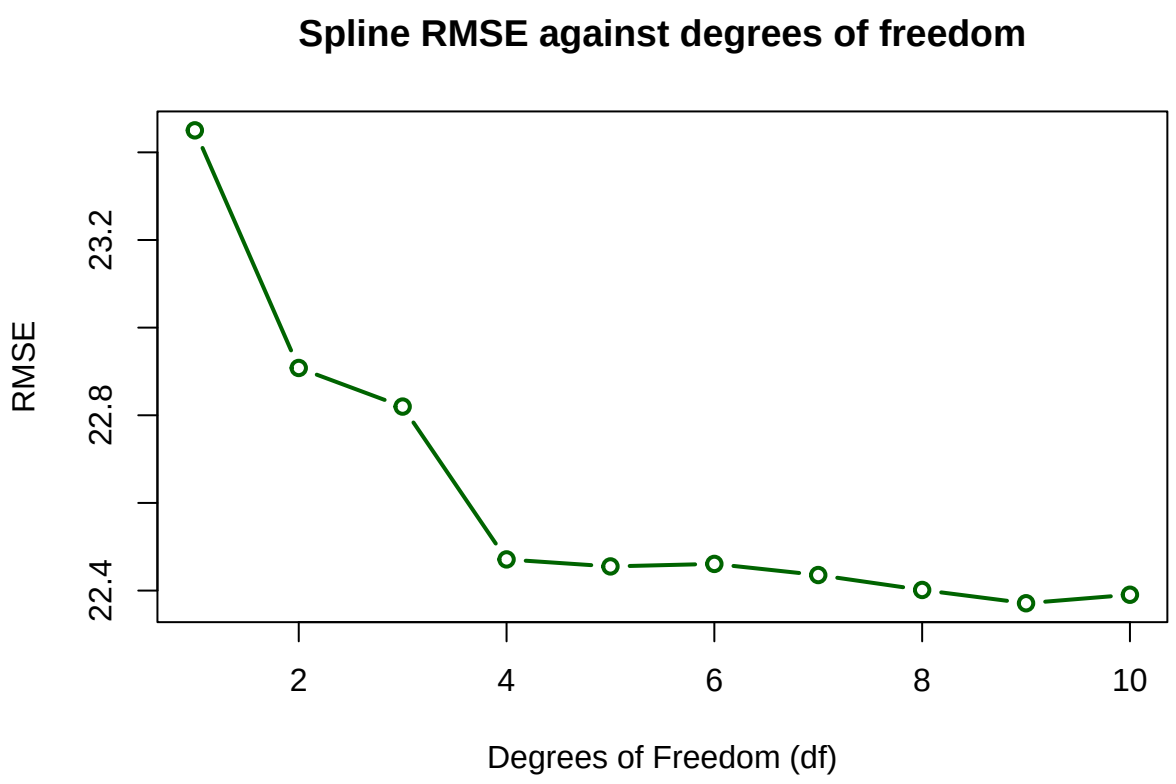
for an appropriate choice of basis functions $b_1, b_2, \ldots, b_{K+d}$. We will fit a natural spline, which adds the constraint that the function of X is required to be linear at the boundary (in the region where X is smaller than the smallest knot, or larger than the largest knot). This produces more stable estimates at the boundaries.

We can use a 10-fold cross-validation to determine how many degrees of freedom is optimal. Below, we print the RMSE on the training set for $df = 1, 2, \ldots, 10$ respectively.

```
#use 10-fold CV to determine optimal degrees of freedom
set.seed(1)
spline_rmse<-c()

#try out df=1,2,...,10
for(i in seq(10)){
spl_fit <- train(as.formula(paste0("critical_temp ~ ns(wtd_std_ThermalConductivity, df=", i, ")")),
                    data = regtrain,
                    method = "lm",
                    trControl = trainControl(method = "cv", number = 10),
                    na.action=na.omit)
pred_dum <- predict(spl_fit, newdata = regtest)
spline_rmse[i]<-RMSE(pred_dum, regtest$critical_temp)
}

#print and plot rmse for spline with df=1,2,...,10
spline_rmse
```

```
##  [1] 23.45015 22.90794 22.81984 22.47114 22.45514 22.46087 22.43551 22.40143
##  [9] 22.37113 22.39050
```

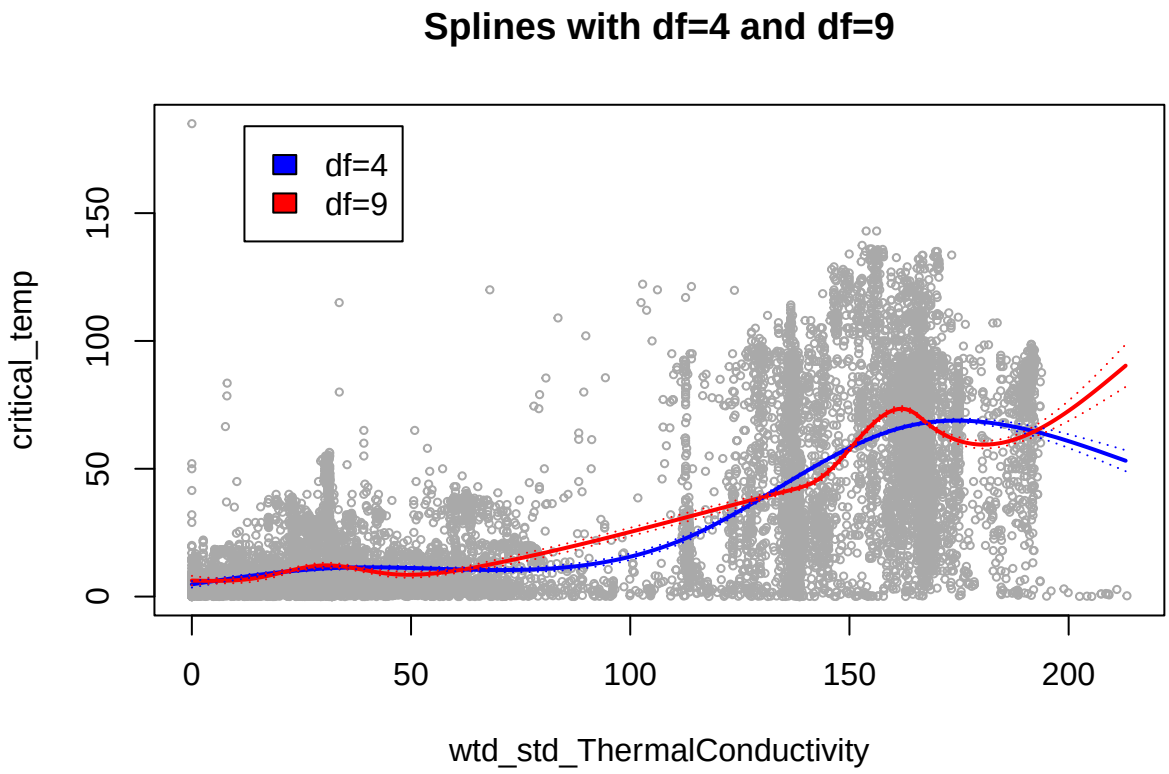
```
plot(seq(10), spline_rmse, type = "b",
     xlab = "Degrees of Freedom (df)", ylab = "RMSE", lwd=2, col="darkgreen", main="Spline RMSE against
```



The optimal RMSE using a natural spline is 22.37, which occurs when $df = 9$. While the spline with $df = 9$ degrees of freedom performs best in terms of RMSE, the spline with only $df = 4$ degrees of freedom performs essentially just as well. We can plot both of them on the same scatterplot:

```
#produce the fits
fitsp4<- lm(critical_temp~ns(wtd_std_ThermalConductivity, df=4), data=regtrain)
fitsp9<- lm(critical_temp~ns(wtd_std_ThermalConductivity, df=9), data=regtrain)

#plot the fit
splims <- range(regtrain$wtd_std_ThermalConductivity)
sp.grid <- seq(from = splims[1], to = splims[2])
predsp4 <- predict(fitsp4, newdata = list(wtd_std_ThermalConductivity = sp.grid), se = TRUE)
se.bansp4 <- cbind(predsp4$fit + 2 * predsp4$se.fit,
    predsp4$fit - 2 * predsp4$se.fit)
predsp9 <- predict(fitsp9, newdata = list(wtd_std_ThermalConductivity = sp.grid), se = TRUE)
se.bansp9 <- cbind(predsp9$fit + 2 * predsp9$se.fit,
    predsp9$fit - 2 * predsp9$se.fit)

plot(regtrain$wtd_std_ThermalConductivity, regtrain$critical_temp, xlim = splims, cex = .5,
     col = "darkgrey", xlab="wtd_std_ThermalConductivity", ylab="critical_temp")
title("Splines with df=4 and df=9")
lines(sp.grid, predsp4$fit, lwd = 2, col = "blue")
lines(sp.grid, predsp9$fit, lwd = 2, col = "red")
matlines(sp.grid, se.bansp4, lwd = 1, col = "blue", lty = 3)
matlines(sp.grid, se.bansp9, lwd = 1, col = "red", lty = 3)
legend(12, 184, legend=c("df=4", "df=9"),
```

```
       fill = c("blue","red"))
```



GAMs

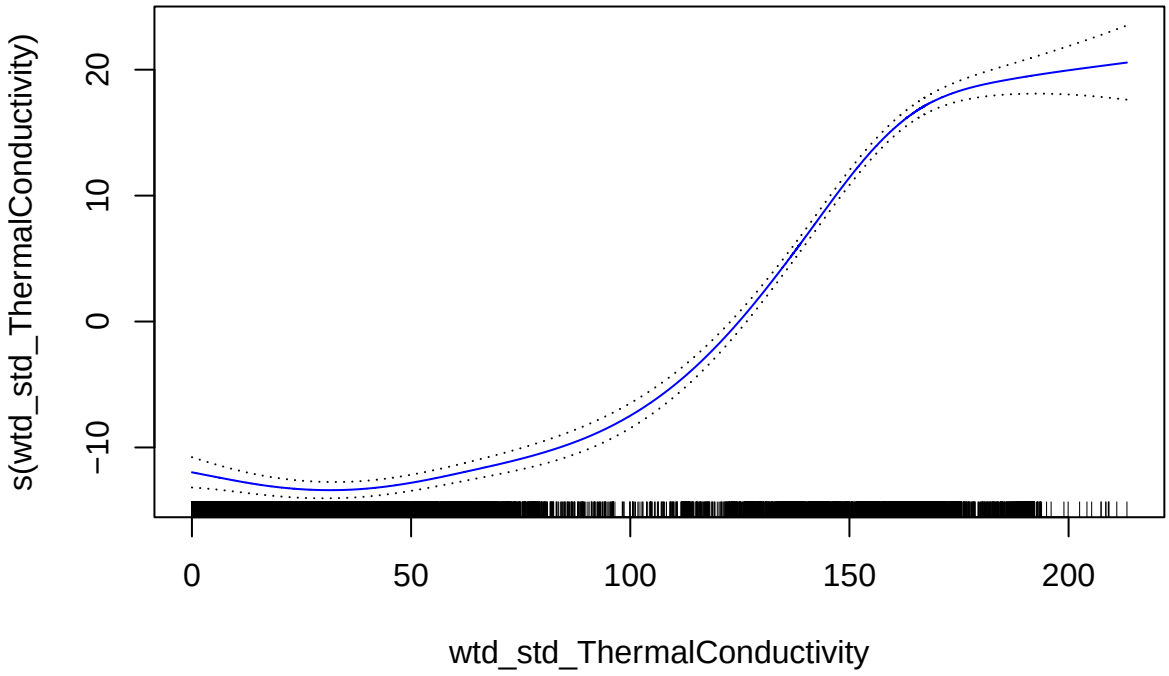
Each of the methods we have employed so far involve modelling Y based on a function of a single predictor X. If we want to flexibly predict Y on the basis of several predictors $X_1, X_2, \ldots, X_p$, we use *generalized additive models (GAMs)*, which replace each linear component $\beta_j x_{ij}$ in OLS with a smooth, nonlinear function $f_j(x_{ij})$. We can write this model as

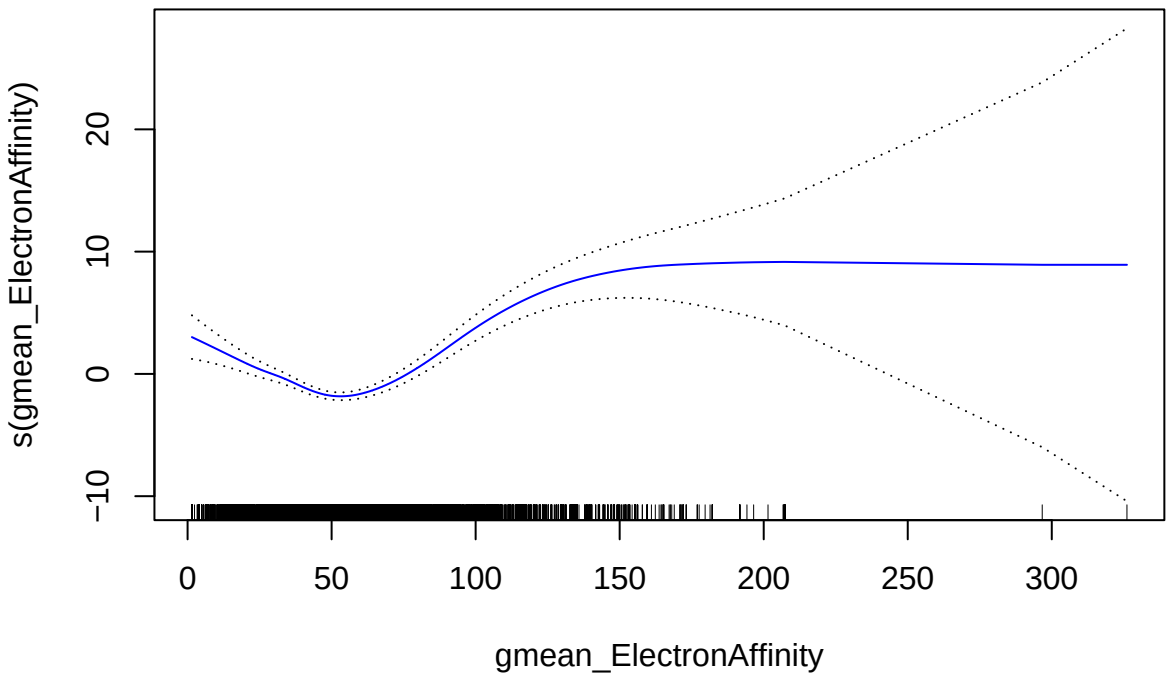
$$\begin{aligned}\hat{y}_i &= \beta_0 + f_1(x_{i1}) + f_2(x_{i2}) + \cdots + f_p(x_{ip}) + \epsilon_i \\ &= \beta_0 + \sum_{j=1}^{p} f_j(x_{ij}) + \epsilon_i.\end{aligned}$$

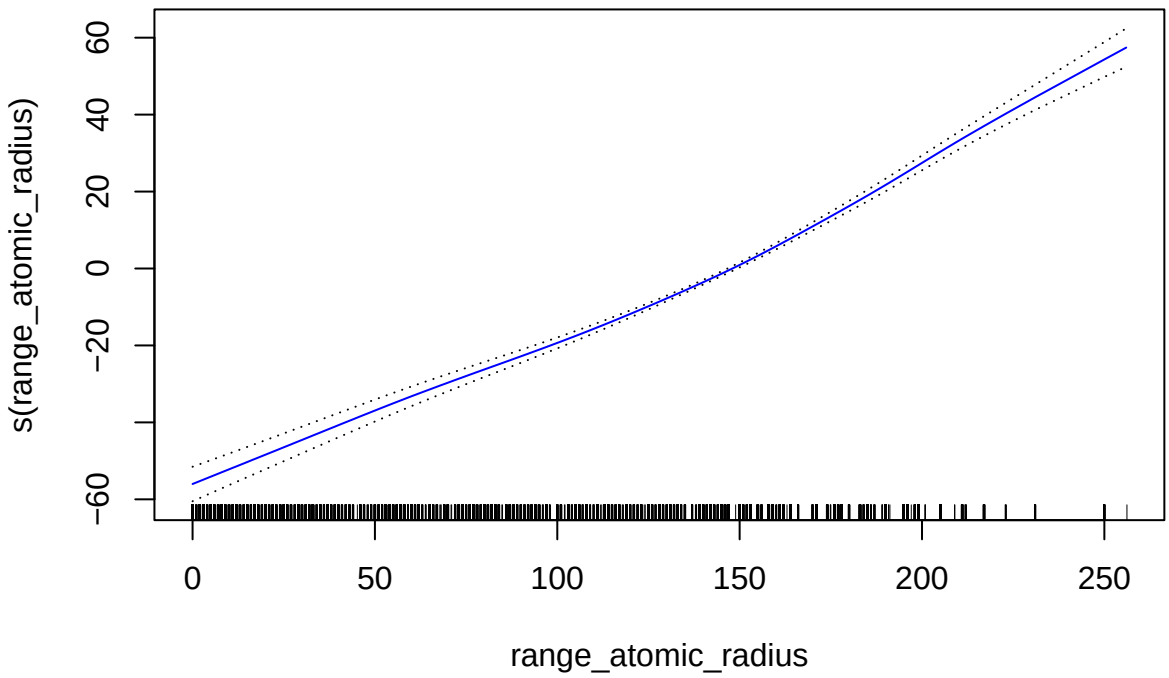
We can now include more predictors from our initial forward stepwise selection. We will fit a model based on the first eight variables chosen: `wtd_std_ThermalConductivity`, `gmean_ElectronAffinity`, `range_atomic_radius`, `std_atomic_radius`, `entropy_ElectronAffinity`, `wtd_gmean_ElectronAffinity`, `wtd_std_Valence`, and `wtd_mean_ElectronAffinity`. These are all quantitative variables, so we use the `s()` function from the `gam` library to specify that we want to fit a smoothing spline to each of them.

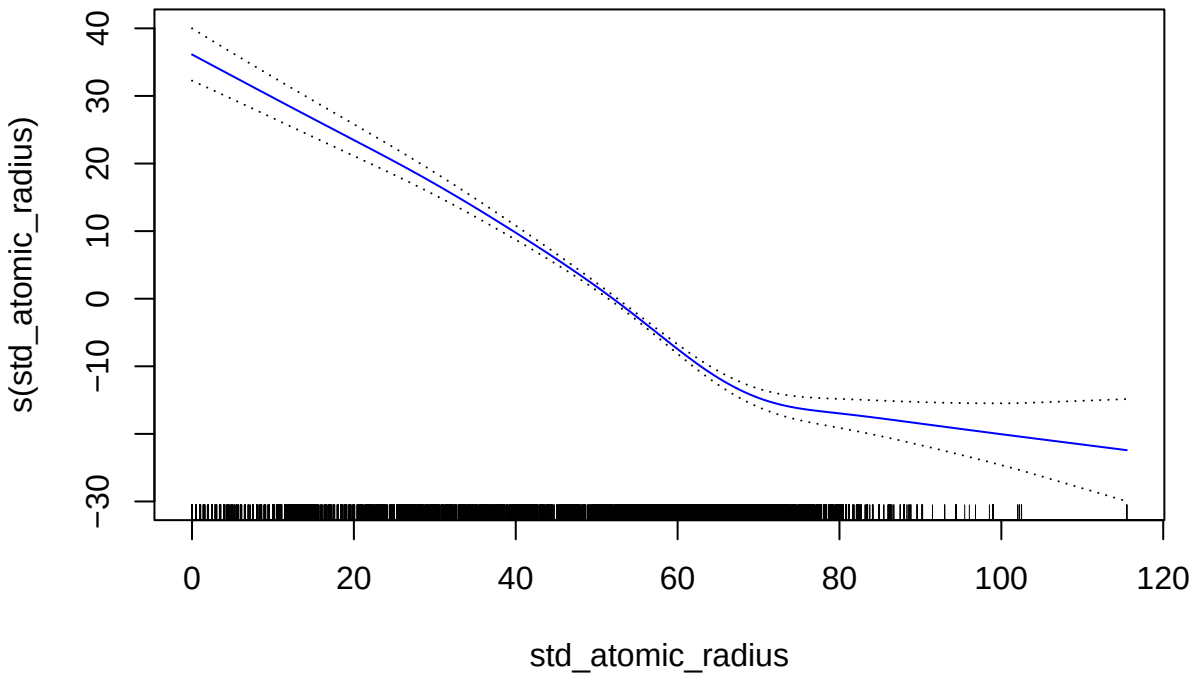
```
#fit gam model using the selected predictors
supgam <- gam(critical_temp~s(wtd_std_ThermalConductivity)+s(gmean_ElectronAffinity)+
                s(range_atomic_radius)+s(std_atomic_radius)+s(entropy_ElectronAffinity)+
                s(wtd_gmean_ElectronAffinity)+s(wtd_std_Valence)+s(wtd_mean_ElectronAffinity),
              data=regtrain)
```

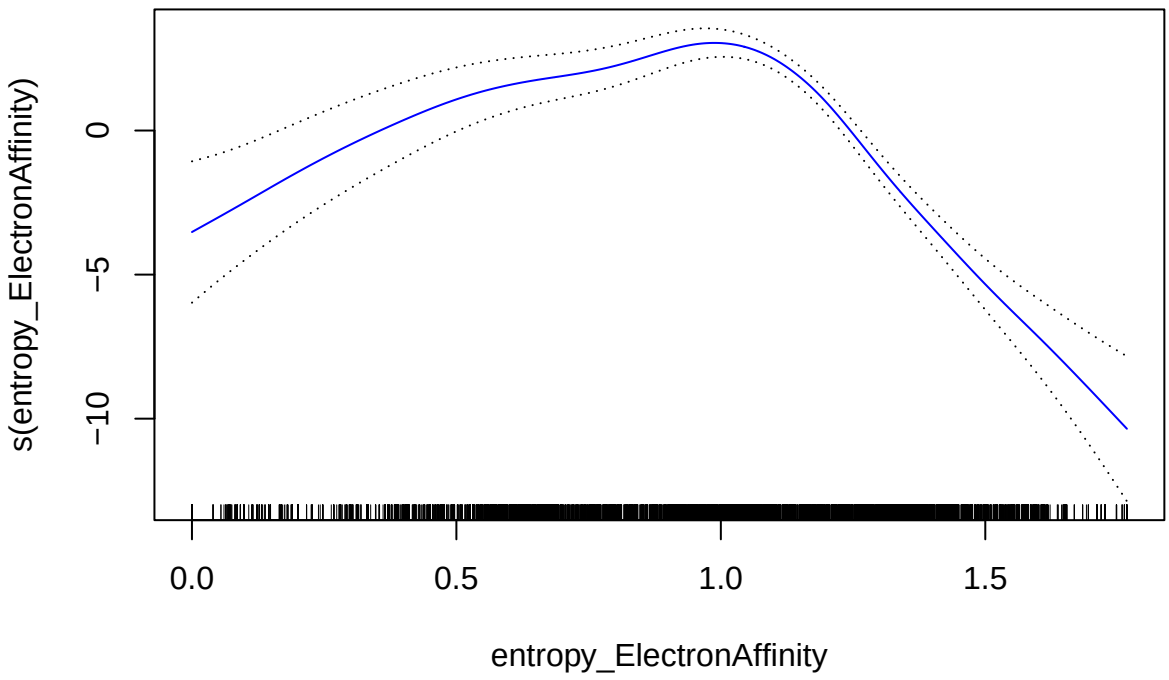
```r
#plot the fit
plot(supgam, se = TRUE, col = "blue")
```

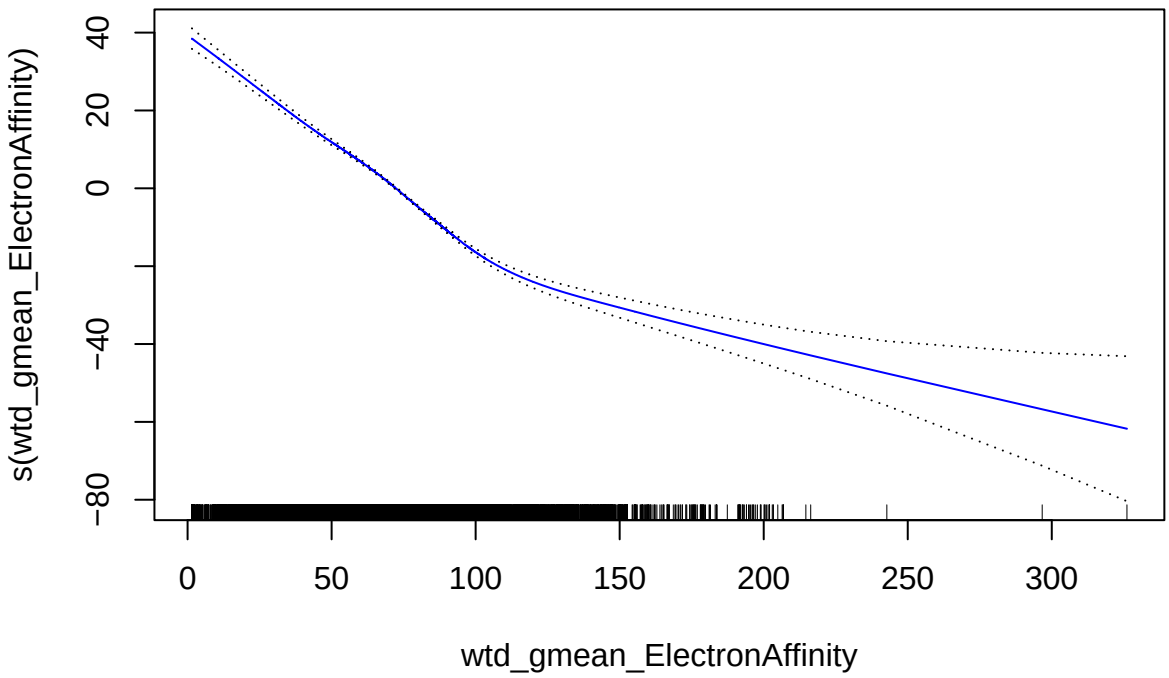


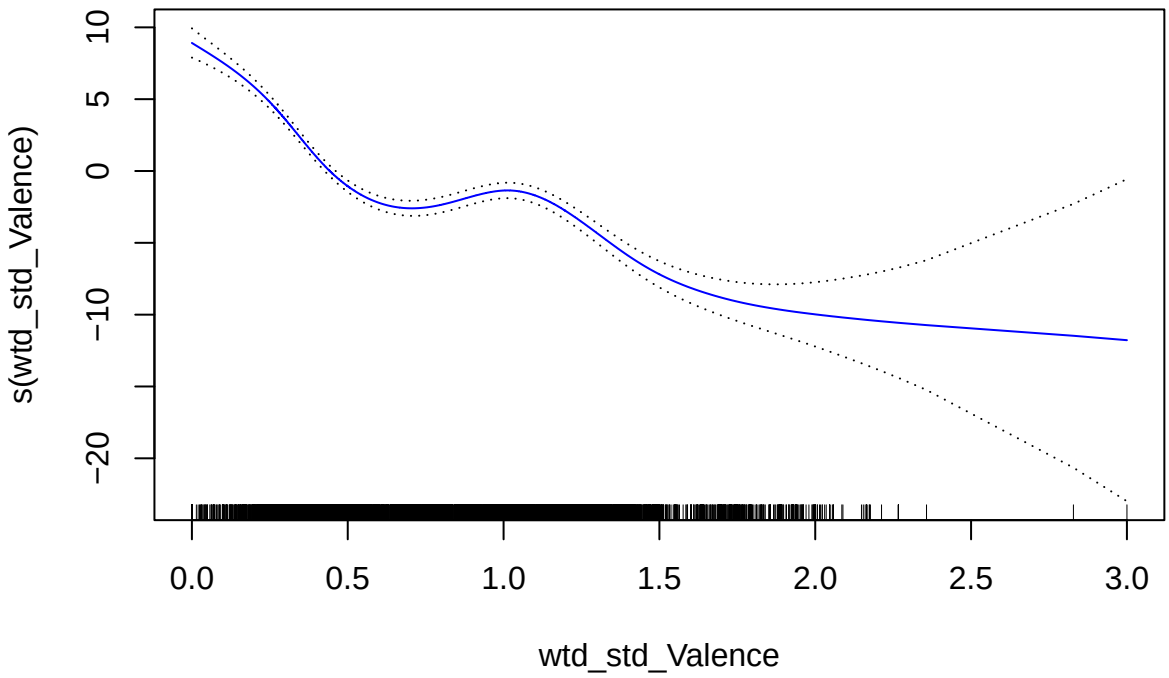

s(gmean_ElectronAffinity)
−10
0
10
20
0
50
100
150
200
250
300
gmean_ElectronAffinity

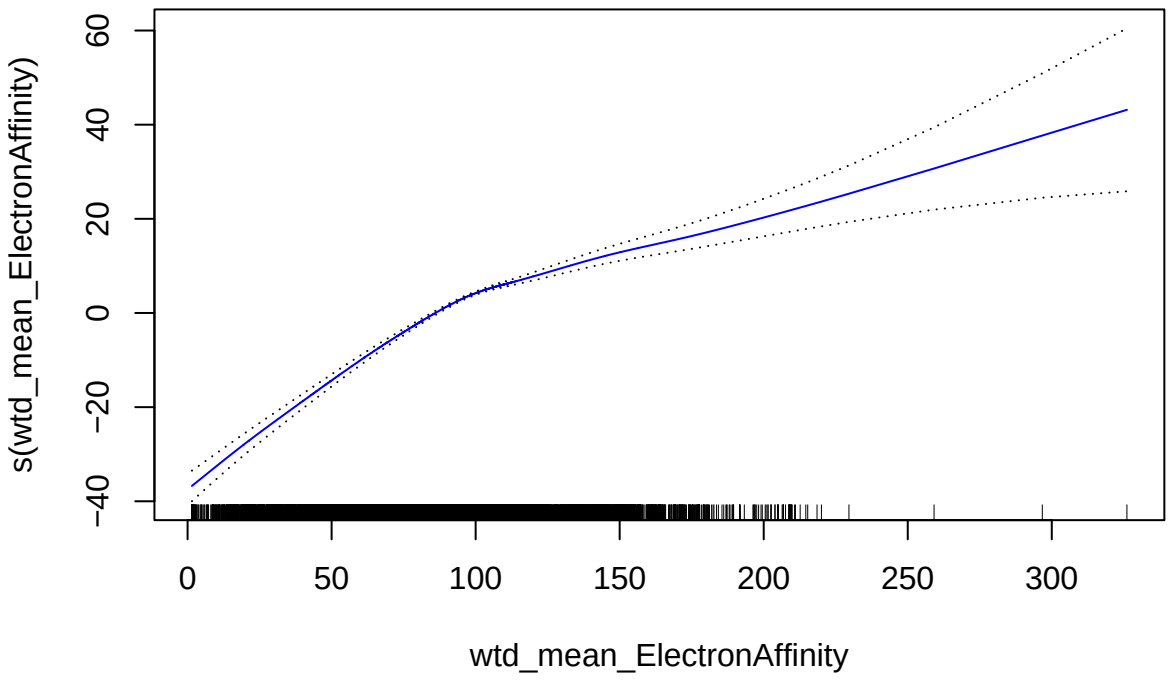

s(range_atomic_radius)
60
40
20
0
−20
−60
0
50
100
150
200
250
range_atomic_radius


s(std_atomic_radius)
40
30
20
10
0
−10
−30
0
20
40
60
80
100
120
std_atomic_radius


s(entropy_ElectronAffinity)
0
−5
−10
0.0
0.5
1.0
1.5
entropy_ElectronAffinity


s(wtd_gmean_ElectronAffinity)
40
20
0
−40
−80
0
50
100
150
200
250
300
wtd_gmean_ElectronAffinity


s(wtd_std_Valence)
10
5
0
−10
−20
0.0
0.5
1.0
1.5
2.0
2.5
3.0
wtd_std_Valence



We can consult the ANOVA to see which predictors are significant, and to see which predictors have a significantly nonlinear relationship with the response. Looking at the plots above, it appears that every variable's relationship with the response is somewhat nonlinear.

```
summary(supgam)
```

```
##
## Call: gam(formula = critical_temp ~ s(wtd_std_ThermalConductivity) +
##     s(gmean_ElectronAffinity) + s(range_atomic_radius) + s(std_atomic_radius) +
##     s(entropy_ElectronAffinity) + s(wtd_gmean_ElectronAffinity) +
##     s(wtd_std_Valence) + s(wtd_mean_ElectronAffinity), data = regtrain)
## Deviance Residuals:
##      Min       1Q   Median       3Q      Max
## -82.4742 -10.5335   0.2765  11.3777 193.7973
##
## (Dispersion Parameter for gaussian family taken to be 359.8937)
##
##     Null Deviance: 18745770 on 15947 degrees of freedom
## Residual Deviance: 5727708 on 15915 degrees of freedom
## AIC: 139160.3
##
## Number of Local Scoring Iterations: NA
##
## Anova for Parametric Effects
##                                  Df  Sum Sq Mean Sq  F value    Pr(>F)
## s(wtd_std_ThermalConductivity)    1 9559287 9559287 26561.42 < 2.2e-16 ***
```

```
## s(gmean_ElectronAffinity)          1  742656  742656  2063.54 < 2.2e-16 ***
## s(range_atomic_radius)             1  564147  564147  1567.54 < 2.2e-16 ***
## s(std_atomic_radius)               1  214477  214477   595.95 < 2.2e-16 ***
## s(entropy_ElectronAffinity)        1   71842   71842   199.62 < 2.2e-16 ***
## s(wtd_gmean_ElectronAffinity)      1  225487  225487   626.54 < 2.2e-16 ***
## s(wtd_std_Valence)                 1   56550   56550   157.13 < 2.2e-16 ***
## s(wtd_mean_ElectronAffinity)       1  168246  168246   467.49 < 2.2e-16 ***
## Residuals                      15915 5727708     360
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Anova for Nonparametric Effects
##                                Npar Df  Npar F     Pr(F)
## (Intercept)
## s(wtd_std_ThermalConductivity)       3 250.827 < 2.2e-16 ***
## s(gmean_ElectronAffinity)            3  56.709 < 2.2e-16 ***
## s(range_atomic_radius)               3  65.474 < 2.2e-16 ***
## s(std_atomic_radius)                 3  91.865 < 2.2e-16 ***
## s(entropy_ElectronAffinity)          3  95.577 < 2.2e-16 ***
## s(wtd_gmean_ElectronAffinity)        3 137.903 < 2.2e-16 ***
## s(wtd_std_Valence)                   3 110.712 < 2.2e-16 ***
## s(wtd_mean_ElectronAffinity)         3  89.278 < 2.2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The Anova for Parametric Effects tells us that every predictor we included is extremely significant, which shows that our stepwise forward selection did its job. Meanwhile, the Anova for Nonparametric Effects indicates that every predictor has a very significantly nonlinear relationship with the response, so the GAM is appropriate to model these complex relationships that linear regression would fail to capture. Finally, we evaluate our GAM on the test set.

```
#evaluate the model on the test set
gampred <- predict(supgam, newdata = regtest)
paste("RMSE for GAM fit: ", sqrt(mean((gampred - regtest$critical_temp)^2)))
```

```
## [1] "RMSE for GAM fit:  19.0703518619506"
```

As expected, with an RMSE of 19.07 the GAM significantly outperforms both the piecewise polynomial model and the spline model. This is not surprising as each of the latter models was only able to harness the predictive power of a single variable, whereas the GAM used eight predictors.

III. Tree-Based Models

Tree-based models involve segmenting the feature space into several smaller regions through a series of "splits" or binary partitions, then classifying new observations based on the mean or majority vote of the region in which the observation lies. Trees are simple to understand and easy to interpret, and can be flexibly applied to regression and classification problems. What's more, their predictive accuracy can be greatly improved by combining many trees, which we will observe by evaluating four different methods via a train-test split and a cross-validation on the training set.

Regression Tree

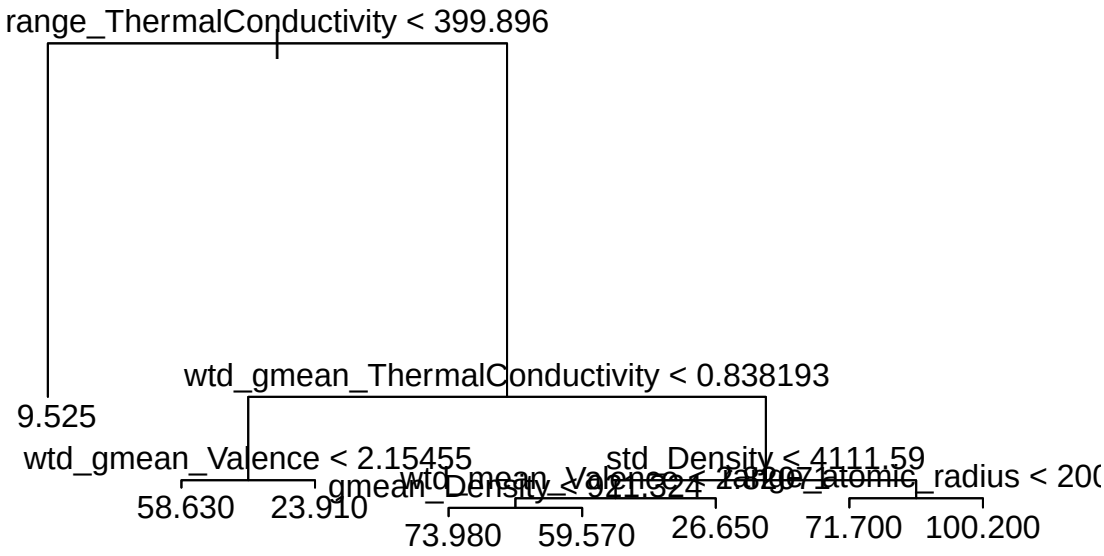
A *decision tree* is a series of splits applied to the data based on *splitting rules*, such as `mean_Density > 4650`, which partition the observations based on the value of a single predictor. In the context of regression, the model chooses the rule for each split which minimizes the sum of squared errors when we make our $\hat{y}$ the means of the region each observation falls into as decided by the rule. To see this in practice, we fit a regression tree to the same data that we used in the previous section.

```
set.seed(12)

#create regression tree
reg_tree <- tree(critical_temp ~ ., regtrain)
summary(reg_tree)
```

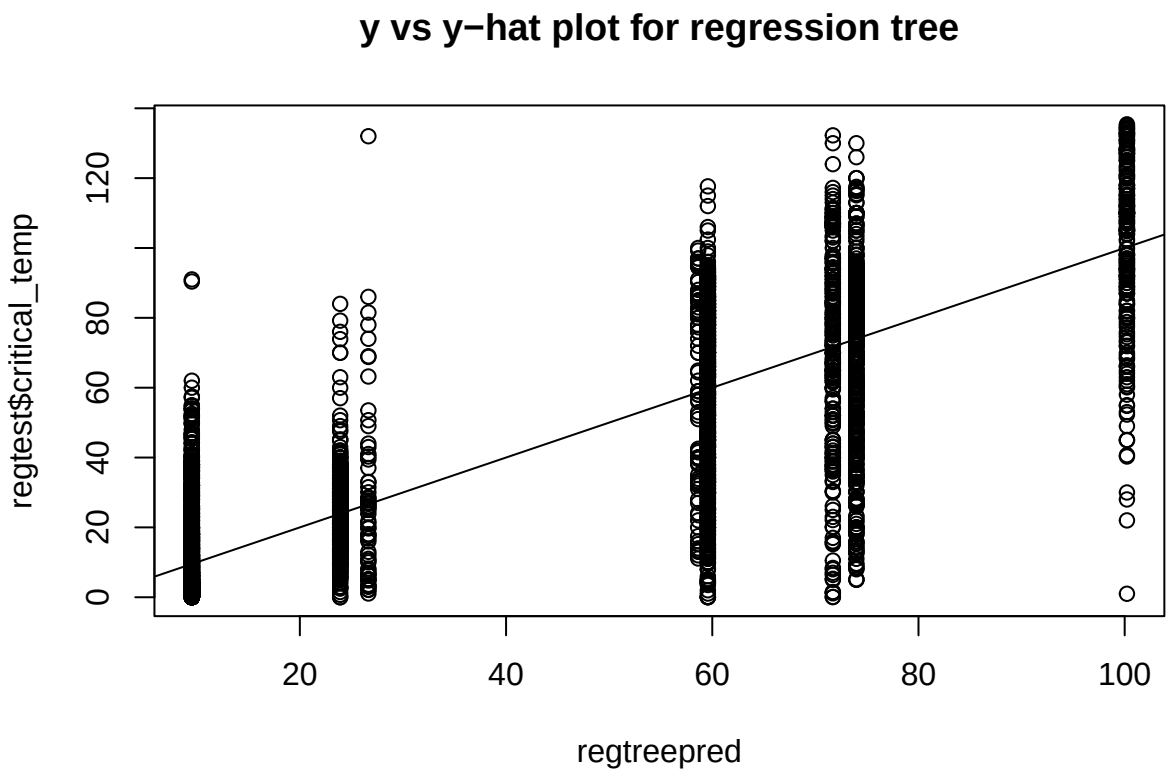
```
##
## Regression tree:
## tree(formula = critical_temp ~ ., data = regtrain)
## Variables actually used in tree construction:
## [1] "range_ThermalConductivity"     "wtd_gmean_ThermalConductivity"
## [3] "wtd_gmean_Valence"             "std_Density"
## [5] "wtd_mean_Valence"              "gmean_Density"
## [7] "range_atomic_radius"
## Number of terminal nodes:  8
## Residual mean deviance:  312.4 = 4980000 / 15940
## Distribution of residuals:
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -90.220  -7.648  -2.150   0.000   9.975 175.500
```

Interestingly, this regression tree and the stepforward selection method both picked out statistics related to thermal conductivity as being highly predictive of critical temperature. We also note that the final tree only created splits based on seven out of 81 possible predictors.

```
#plot the tree
plot(reg_tree)
text(reg_tree, pretty = 0)
```




```
#evaluate performance
regtreepred<-predict(reg_tree, newdata=regtest)
plot(regtreepred, regtest$critical_temp, main="y vs y-hat plot for regression tree")
abline(0,1)
```




```
paste("RMSE for unpruned tree: ", sqrt(mean((regtreepred-regtest$critical_temp)^2)))
```

```
## [1] "RMSE for unpruned tree:  18.02560678578"
```

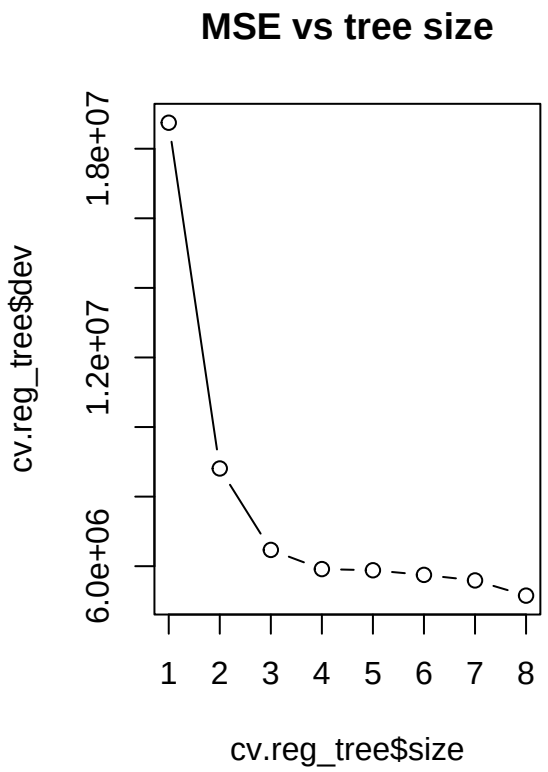
Observe that the model made its first split based on the variable `range_ThermalConductivity`, which already resulted in a terminal node on the next layer, so the model predicts that `critical_temp = 9.525` for the superconductors for which `range_ThermalConductivity < 399.896`. The RMSE for the unpruned tree evaluated on the test set is 18.02, so on average the predictions made by our regression tree differ from the observed values in the test set by 18.02 K. Interestingly, this regression tree also outperforms the GAM, which makes sense because the regression tree was able to consider every possible predictor, while the GAM was fitted based on a specified subset of eight predictors.

```
set.seed(334)
cv.reg_tree <- cv.tree(reg_tree)
cv.reg_tree
```

```
## $size
## [1] 8 7 6 5 4 3 2 1
##
## $dev
## [1]  5154255  5591254  5749988  5882715  5917041  6468158  8809171 18747301
##
## $k
## [1]      -Inf  208619.6  236047.9  260011.7  269479.8  507053.9 2345225.4
```

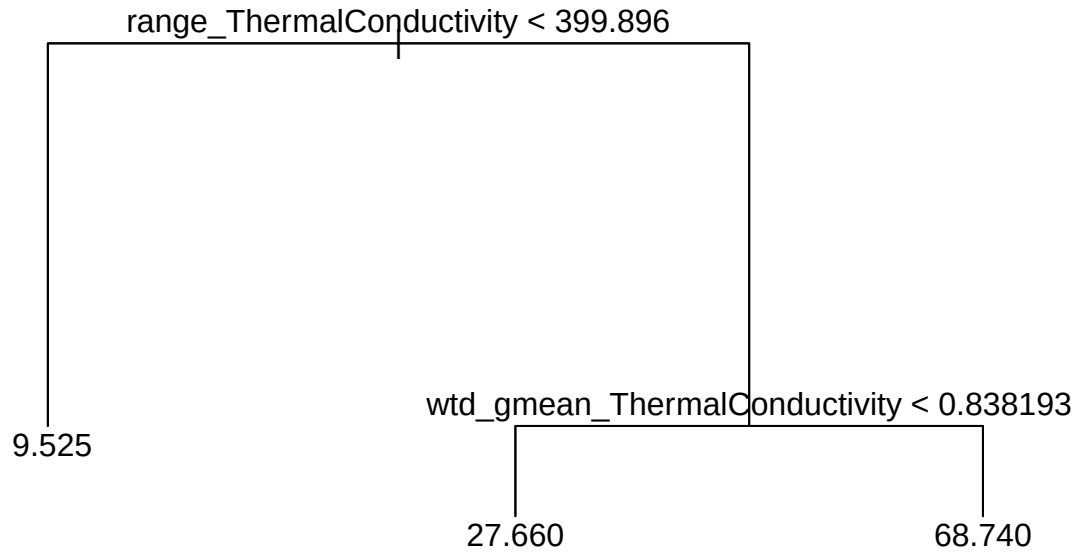
```
## [8] 9938987.7
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
par(mfrow = c(1, 2))
plot(cv.reg_tree$size, cv.reg_tree$dev, type = "b", main="MSE vs tree size")
```



The 10-fold cross-validation selects 8 terminal nodes as optimal, which is exactly what the `tree()` function did without any pruning. Note that, in the context of regression, the deviance reported by the `cv.tree()` function represents the mean squared error (MSE) of each model. We can try pruning with 3 terminal nodes to verify that we get a worse performance on the test set.

```
prune.regtree <- prune.tree(reg_tree, best = 3)
plot(prune.regtree)
text(prune.regtree, pretty = 0)
```




```
#evaluate performance
regtreepredune<-predict(prune.regtree, newdata=regtest)
paste("RMSE for pruned tree: ", sqrt(mean((regtreepredune-regtest$critical_temp)^2)))
```

```
## [1] "RMSE for pruned tree:  20.3546405119529"
```

The RMSE for the pruned tree is 20.35, so the unpruned tree does indeed outperform the pruned tree.

Classification Tree

Applying a decision tree to a classification problem is very similar to what we already did in the regression setting, but now we choose splits based on the classification accuracy associated with each splitting rule. For the purpose of studying classification trees, we employ a data set involving the binary classification of 569 tumor cells as benign or malignant, based on 31 features extracted from visual analysis of images of the tumors. We begin by reading in and cleaning our data, before performing a train-test split.

```
#load in data
cancer <- fread("C:/Users/Theo/Downloads/Cancer_Data.csv",fill=TRUE)
garbage0<-class(as.data.frame(cancer))
cancer<-cancer[,-33]
cancer<-na.omit(cancer)
dum <- factor(ifelse(cancer$diagnosis=="M", "Malignant", "Benign"))
cancer$diagnosis<-dum
colnames(cancer) <- make.names(colnames(cancer))
```

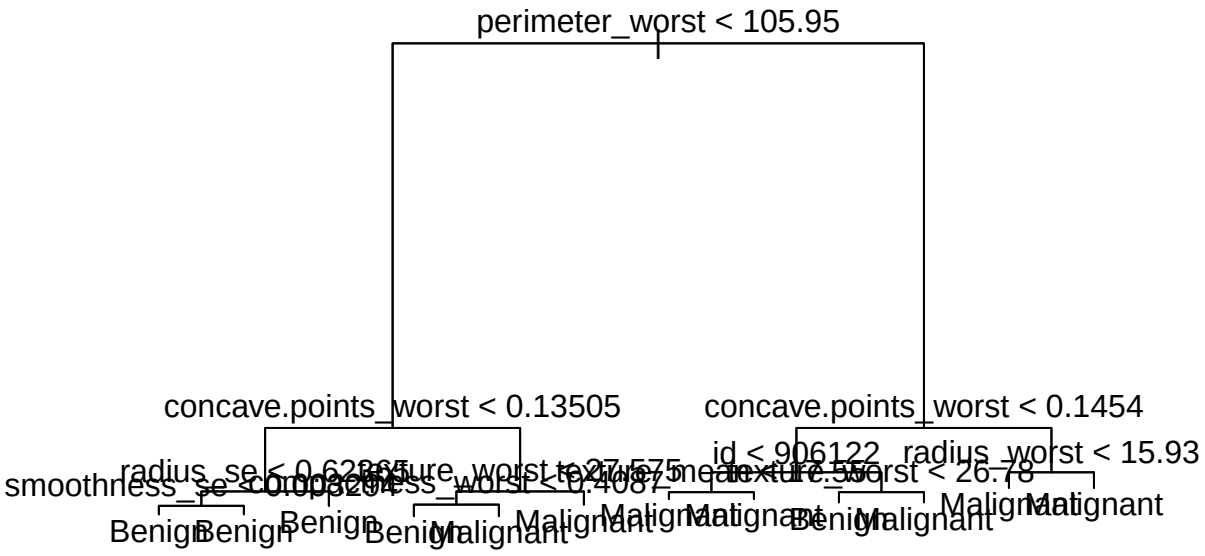
```r
#set a seed to ensure our data is reproducible
set.seed(123)

#create 75% training 25% testing split
clatrainIndex <- createDataPartition(cancer$diagnosis, p = .75, list = FALSE, times = 1)
clatrain <- cancer[clatrainIndex, ]
clatest <- cancer[-clatrainIndex, ]
```

Now, we fit a classification tree to our training data.

```r
#create tree model
cla_tree <- tree(diagnosis~., clatrain)
summary(cla_tree)
```

```
##
## Classification tree:
## tree(formula = diagnosis ~ ., data = clatrain)
## Variables actually used in tree construction:
## [1] "perimeter_worst"      "concave.points_worst" "radius_se"
## [4] "smoothness_se"        "texture_worst"        "compactness_worst"
## [7] "id"                   "texture_mean"         "radius_worst"
## Number of terminal nodes:  12
## Residual mean deviance:  0.09678 = 40.16 / 415
## Misclassification error rate: 0.02576 = 11 / 427
```

```r
plot(cla_tree)
text(cla_tree, pretty=0)
```



We note that the first split is based on `perimeter_worst`, while the two splits on the next layer are both based on `concave.points_worst`, so the model has decided that these two variables are important in predicting whether a mass is benign or malignant. The model has used only nine variables out of a possible 31 to segment the data, and it has 12 terminal nodes. We evaluate its accuracy on the test set.

```
#evaluate full tree model on the test set
cla_tree.pred <- predict(cla_tree, clatest,
    type = "class")
confusionMatrix(cla_tree.pred, clatest$diagnosis)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Benign Malignant
##   Benign        79         3
##   Malignant     10        50
##
##                Accuracy : 0.9085
##                  95% CI : (0.8485, 0.9503)
##     No Information Rate : 0.6268
##     P-Value [Acc > NIR] : 1.905e-14
##
##                   Kappa : 0.8094
##
##  Mcnemar's Test P-Value : 0.09609
##
```

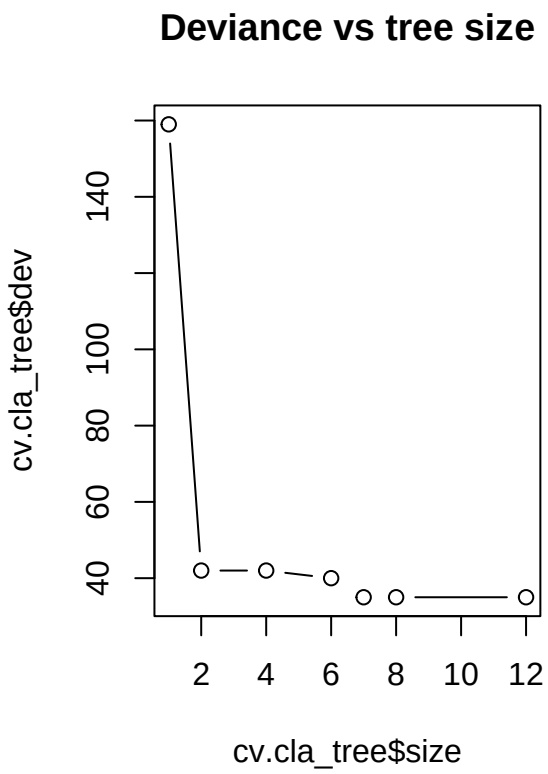
```
##             Sensitivity : 0.8876
##             Specificity : 0.9434
##          Pos Pred Value : 0.9634
##          Neg Pred Value : 0.8333
##              Prevalence : 0.6268
##          Detection Rate : 0.5563
##    Detection Prevalence : 0.5775
##       Balanced Accuracy : 0.9155
##
##        'Positive' Class : Benign
##
```

The single, unpruned classification tree performs well on the test set with 90.85% accuracy. It has higher specificity than sensitivity, so it is more likely to produce a false negative than a false positive. To evaluate whether pruning the tree will improve its performance on the test set, we employ a 10-fold cross-validation.

```
set.seed(21)

#prune tree through 10-fold CV
cv.cla_tree <- cv.tree(cla_tree, FUN = prune.misclass)
cv.cla_tree
```

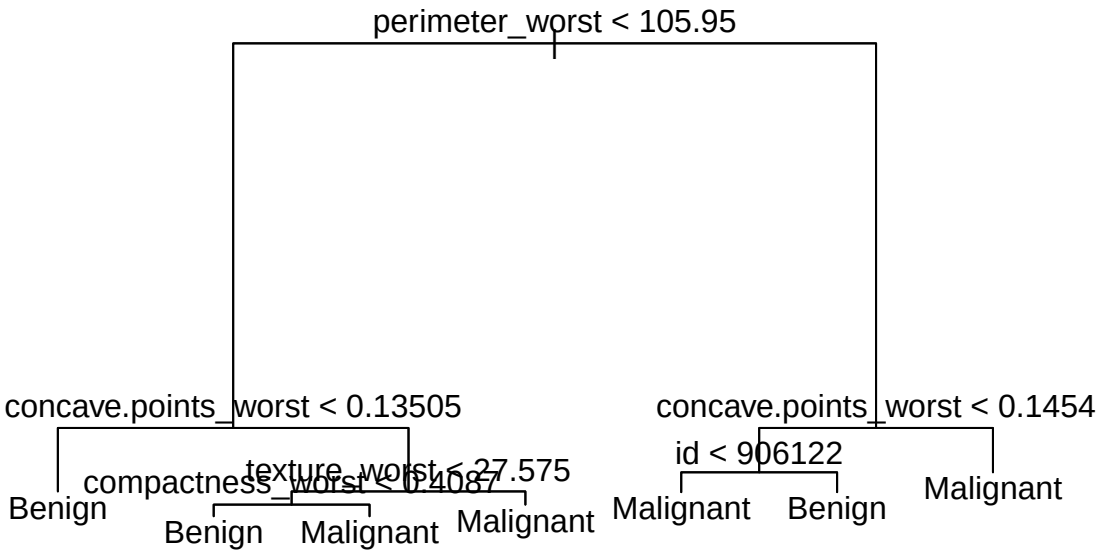
```
## $size
## [1] 12  8  7  6  4  2  1
##
## $dev
## [1]  35  35  35  40  42  42 159
##
## $k
## [1]  -Inf   0.0   1.0   3.0   4.0   4.5 127.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
#plot results of CV
par(mfrow = c(1, 2))
plot(cv.cla_tree$size, cv.cla_tree$dev, type = "b", main="Deviance vs tree size")
```



The cross-validation suggests that pruning the tree so it has only seven terminal nodes is optimal; we do so and reevaluate its performance.

```
#prune tree and plot it
prune.cla_tree <- prune.misclass(cla_tree, best = 7)
plot(prune.cla_tree)
text(prune.cla_tree, pretty = 0)
```




```
#evaluate pruned tree on test set
prunecla.pred <- predict(prune.cla_tree, clatest,
    type = "class")
confusionMatrix(prunecla.pred, clatest$diagnosis)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Benign Malignant
##   Benign        81         5
##   Malignant      8        48
##
##                Accuracy : 0.9085
##                  95% CI : (0.8485, 0.9503)
##     No Information Rate : 0.6268
##     P-Value [Acc > NIR] : 1.905e-14
##
##                   Kappa : 0.8065
##
##  Mcnemar's Test P-Value : 0.5791
##
##             Sensitivity : 0.9101
##             Specificity : 0.9057
##          Pos Pred Value : 0.9419
##          Neg Pred Value : 0.8571
##              Prevalence : 0.6268
```

```
##          Detection Rate : 0.5704
##    Detection Prevalence : 0.6056
##       Balanced Accuracy : 0.9079
##
##        'Positive' Class : Benign
##
```

We have the exact same classification accuracy, but a significantly simpler and more interpretable tree. We therefore prefer the pruned tree as it is more *parsimonious*. Note also that the first splits of the pruned tree are based on the exact same variables as the first splits of the unpruned tree, namely `perimeter_worst` and `concave.points_worst`.

Random Forest

While decision trees have many key advantages, such as their easy interpretability, they also suffer from a high sensitivity to the data on which they are trained. To lower variance, and thereby improve predictive accuracy, we can aggregate many decision trees and average over them. To do so, we use a procedure called *bagging*, or bootstrap aggregating.

To perform bagging, we generate B different bootstrapped data sets from our training set. We then train our method (in this case, we create a decision tree) on the bth bootstrapped data set to get $\hat{f}^{*b}(x)$, then average over all the bootstrapped data sets, yielding

$$\hat{f}_{bag}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{*b}(x).$$

Now, we want to fit a *random forest* model, which is essentially the same as the above but with a crucial caveat. We perform bagging, but when building each decision tree, we only consider a random sample of $m < p$ predictors each time a split in the tree is decided. This small tweak decorrelates each of the trees we produce, substantially reducing variance when we average over them. In general, when building classification trees (as we are here), we would default to $m = \sqrt{p}$.

Within the `randomForest` library, we can control B with the `ntree` parameter; its default value is 500, which should be sufficient for our data set. A higher number of trees will perform better, but we should experience diminishing returns once our accuracy rate starts to converge, and producing more trees linearly increases the computation we need. So, we will be optimizing the hyperparameter m by finding the optimal value of the parameter `mtry` through a 10-fold cross-validation, repeated three times.

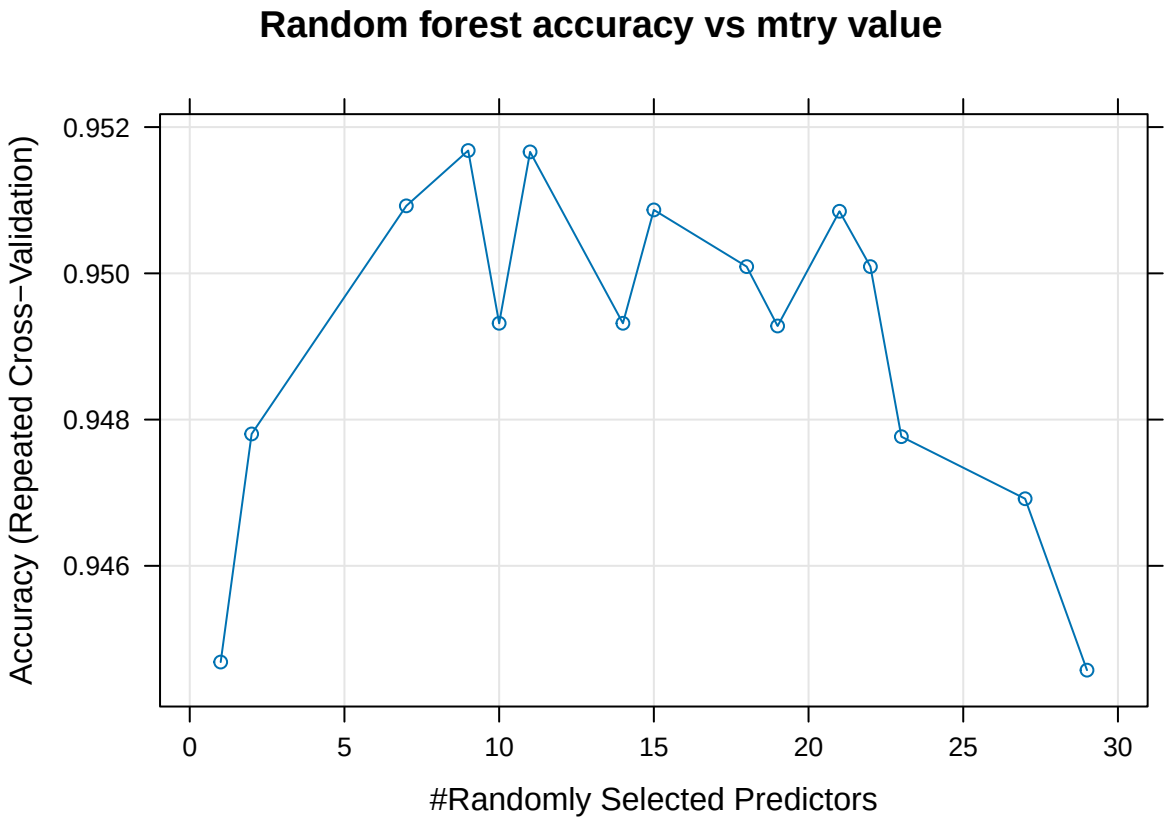
```
#create train control: 10-fold cv, repeated 3 times
control <- trainControl(method='repeatedcv',
                        number=10,
                        repeats=3,
                        search = 'random')

#set seed for reproducibility
set.seed(1)

#have caret generate 20 mtry values with tuneLength = 20
rf_random <- train(diagnosis ~ .,
                   data = clatrain,
                   method = 'rf',
                   metric = 'Accuracy',
                   tuneLength  = 20,
```

```
                    trControl = control)
print(rf_random)
```

```
## Random Forest
##
## 427 samples
##  31 predictor
##   2 classes: 'Benign', 'Malignant'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 385, 384, 384, 384, 384, 384, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    1    0.9446844  0.8811436
##    2    0.9478036  0.8882017
##    7    0.9509228  0.8955300
##    9    0.9516796  0.8968376
##   10    0.9493171  0.8921060
##   11    0.9516611  0.8966544
##   14    0.9493171  0.8921167
##   15    0.9508675  0.8951274
##   18    0.9500923  0.8936684
##   19    0.9492802  0.8921278
##   21    0.9508490  0.8952394
##   22    0.9500923  0.8935879
##   23    0.9477667  0.8887125
##   27    0.9469177  0.8870296
##   29    0.9445736  0.8821819
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 9.
```

We observe that the random forest model with $m = 9$ performed the best on the training set. We can also plot the results from our 10-fold cross-validation to visualize how varying m impacts the model's performance in terms of classification accuracy.

```
plot(rf_random, main="Random forest accuracy vs mtry value")
```



Now, we create a random forest model with the optimized value $m = 9$ to see how it will perform on the test set.

```
set.seed(1)

#train the model with mtry=9
rf_final <- randomForest(diagnosis~., data=clatrain, mtry=9)

#evaluate its performance on the test set
rf.pred <- predict(rf_final, clatest,
    type = "class")
confusionMatrix(rf.pred, clatest$diagnosis)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Benign Malignant
##   Benign        86         1
##   Malignant      3        52
##
##                Accuracy : 0.9718
##                  95% CI : (0.9294, 0.9923)
##     No Information Rate : 0.6268
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9402
```

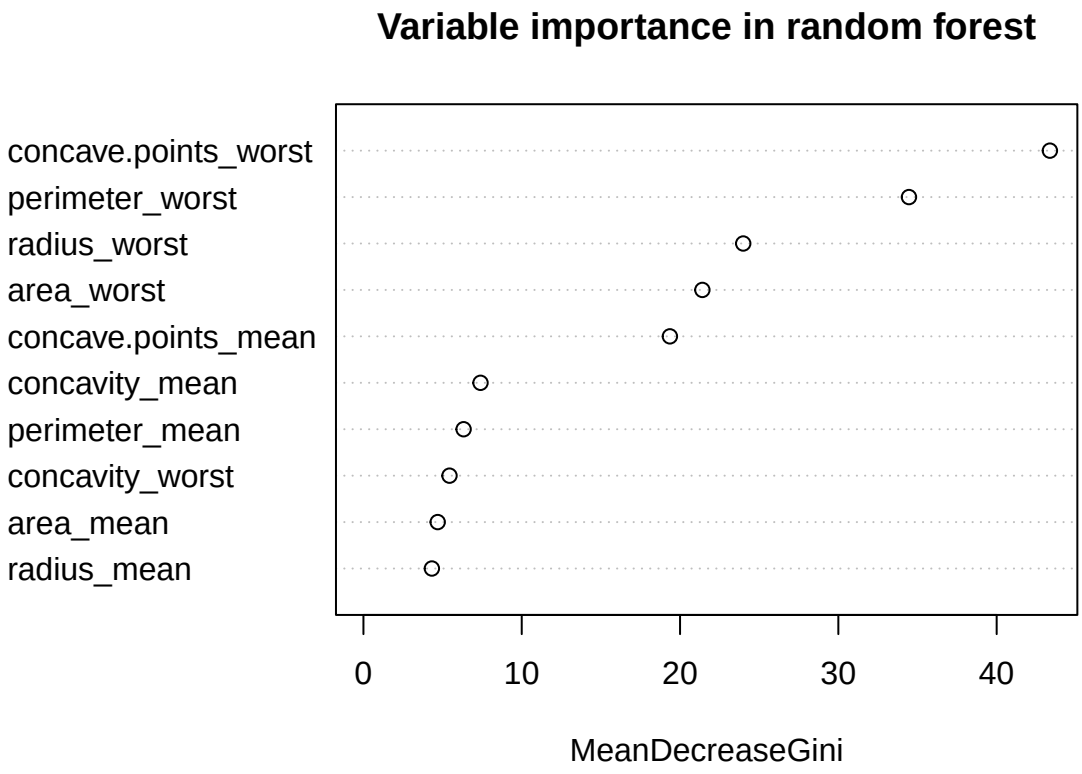
```
##
##  Mcnemar's Test P-Value : 0.6171
##
##             Sensitivity : 0.9663
##             Specificity : 0.9811
##          Pos Pred Value : 0.9885
##          Neg Pred Value : 0.9455
##              Prevalence : 0.6268
##          Detection Rate : 0.6056
##    Detection Prevalence : 0.6127
##       Balanced Accuracy : 0.9737
##
##        'Positive' Class : Benign
##
```

The random forest model with $m = 9$ performed significantly better than the single classification tree did, with a test set accuracy of 96.5%. In the interest of preserving some interpretability, we can also make note of which predictors our random forest model considered the most important.

```
#print and plot importance of variables
importance(rf_final)
```

```
##                          MeanDecreaseGini
## id                              1.3415683
## radius_mean                     4.3229369
## texture_mean                    3.0564668
## perimeter_mean                  6.3263957
## area_mean                       4.6906934
## smoothness_mean                 1.1860756
## compactness_mean                1.0027151
## concavity_mean                  7.3952241
## concave.points_mean            19.3593202
## symmetry_mean                   0.8142512
## fractal_dimension_mean          0.5385081
## radius_se                       1.2447897
## texture_se                      0.7031019
## perimeter_se                    1.0636102
## area_se                         2.6636207
## smoothness_se                   0.6495607
## compactness_se                  0.5617820
## concavity_se                    0.7786859
## concave.points_se               0.4531763
## symmetry_se                     0.6078085
## fractal_dimension_se            1.0962009
## radius_worst                   23.9903056
## texture_worst                   4.2383585
## perimeter_worst                34.4603985
## area_worst                     21.4101616
## smoothness_worst                1.8923728
## compactness_worst               1.3723291
## concavity_worst                 5.4359641
## concave.points_worst           43.3635951
## symmetry_worst                  2.0271120
## fractal_dimension_worst         0.9176506
```

```
varImpPlot(rf_final, n.var=10, main="Variable importance in random forest")
```



Variable importance for a classification task is computed using the mean decrease in Gini index,

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

and is expressed relative to the maximum. We note that the two most important predictors according to this metric are `perimeter_worst` and `concave.points_worst`, followed by the group of three predictors `radius_worst`, `area_worst`, and `concave.points_mean`.

Boosting

We now consider *boosting*, another way to improve the predictive accuracy of decision trees. Boosting is a slow learner, meaning it gradually improves on some model (in this case, a decision tree) which fits to the data only one time, and therefore can suffer from overfitting. Like bagging, boosting involves fitting some large number B of trees and combining them, but unlike bagging, each new tree is grown using information from all of the previously grown trees. The rate at which boosting "learns", i.e. how much we allow each new tree to influence the next tree, is controlled by the shrinkage parameter λ.

Boosting involves multiple hyperparameters which we have to optimize. Unlike when we considered random forest models, taking too large a value of B can actually lead to overfitting in the context of boosting, so we have to account for this in choosing the `n.trees` parameter value. We also optimize the shrinkage parameter `shrinkage` and the parameter `n.minobsinnode`, which sets the minimum number of observations in each terminal node. Finally, we take into account the complexity of each tree by modifying the maximum

tree depth value `interaction.depth`. Note that we set `distribution=bernoulli` as we are dealing with a binary classification problem.

```
#create grid of hyperparameter values to try
gbm_grid <- expand.grid(
  n.trees = c(100, 200, 300, 400, 500),
  interaction.depth = c(1, 2, 3, 5),
  shrinkage = c(0.01, 0.1, 0.2),
  n.minobsinnode = c(3, 5, 10) #try a smaller value bc not very large training sample, default is 10
)

#set seed for reproducibility
set.seed(99)

#train model using cv with caret
gbm_caret <- train(
  diagnosis ~ .,
  data = clatrain,
  method = "gbm",
  distribution = "bernoulli",
  trControl = trainControl(method = "cv", number = 10, verboseIter = FALSE),
  verbose = FALSE,
  metric = 'Accuracy',
  tuneGrid = gbm_grid
)

#print the optimal hyperparameter values
print(gbm_caret$bestTune)
```

```
##     n.trees interaction.depth shrinkage n.minobsinnode
## 124     400                 1       0.2              3
```

We find through 10-fold cross-validation that the optimal hyperparameter values are those above. We now evaluate the performance of this optimal boosted model on the test set.

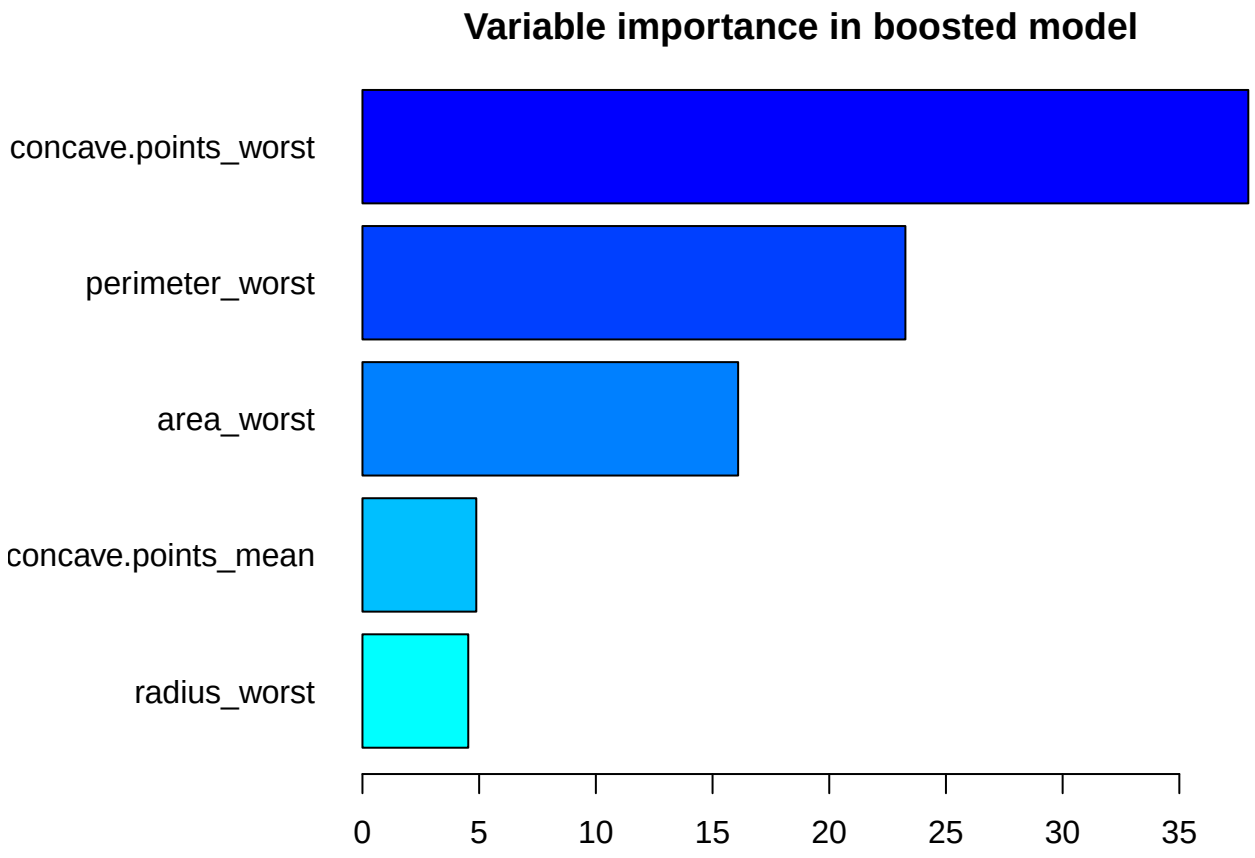
```
#evaluate performance on test set
gbm.pred <- predict(gbm_caret, clatest)
confusionMatrix(gbm.pred, clatest$diagnosis)
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Benign Malignant
##   Benign        87         1
##   Malignant      2        52
##
##                Accuracy : 0.9789
##                  95% CI : (0.9395, 0.9956)
##     No Information Rate : 0.6268
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.955
##
```

```
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9775
##             Specificity : 0.9811
##          Pos Pred Value : 0.9886
##          Neg Pred Value : 0.9630
##              Prevalence : 0.6268
##          Detection Rate : 0.6127
##    Detection Prevalence : 0.6197
##       Balanced Accuracy : 0.9793
##
##        'Positive' Class : Benign
##
```

Our optimal model with boosting has an accuracy of 97.9% on the test set, which outperforms both random forest and the individual classification tree. Finally, we can see how the `gbm()` model ranked the importance of our predictors.

```
#plot variable importance for 5 most important variables
par(mar=c(2,9,2,0.2))
summary(gbm_caret$finalModel, cBars=5, las=1, cex.lab=0.75, main="Variable importance in boosted model")
```




```
##                                       var      rel.inf
## concave.points_worst       concave.points_worst 3.795529e+01
## perimeter_worst                 perimeter_worst 2.326359e+01
```

```
## area_worst                          area_worst 1.609581e+01
## concave.points_mean         concave.points_mean 4.877795e+00
## radius_worst                       radius_worst 4.534162e+00
## texture_worst                     texture_worst 2.053456e+00
## symmetry_worst                   symmetry_worst 1.245286e+00
## texture_mean                       texture_mean 1.080372e+00
## texture_se                           texture_se 1.036260e+00
## symmetry_se                         symmetry_se 1.006388e+00
## id                                           id 9.412114e-01
## area_se                                 area_se 8.601714e-01
## area_mean                             area_mean 6.931458e-01
## concavity_se                       concavity_se 6.920872e-01
## fractal_dimension_se       fractal_dimension_se 6.640664e-01
## smoothness_se                     smoothness_se 5.559681e-01
## smoothness_worst               smoothness_worst 5.391404e-01
## concavity_worst                 concavity_worst 5.032195e-01
## concavity_mean                   concavity_mean 2.936766e-01
## fractal_dimension_worst fractal_dimension_worst 2.598640e-01
## compactness_worst             compactness_worst 2.529717e-01
## radius_se                             radius_se 1.985758e-01
## symmetry_mean                     symmetry_mean 1.669452e-01
## fractal_dimension_mean   fractal_dimension_mean 7.410337e-02
## compactness_se                   compactness_se 7.147619e-02
## concave.points_se             concave.points_se 6.141065e-02
## smoothness_mean                 smoothness_mean 1.206955e-02
## perimeter_se                       perimeter_se 8.268929e-03
## perimeter_mean                   perimeter_mean 3.005186e-03
## radius_mean                         radius_mean 2.094656e-04
## compactness_mean                 compactness_mean 0.000000e+00
```

The ranking of each predictor's importance is almost identical to that of random forest when it comes to the most important predictors, as each ranking has the same top five most important variables.

IV. Conclusion

We considered five different models to handle the same regression task of predicting the critical temperature of superconductors, and evaluated each in terms of RMSE. Although the GAM performed the best in predicting the test set out of all the OLS-based methods, it was outperformed by the regression tree. With that being said, the GAM also had to use a specified subset of predictors while the regression tree was optimized over all predictors, so this result isn't surprising.

In the classification context, the method that achieved the best accuracy in predicting the test set was gradient boosting, followed by random forests, both of which significantly outperformed the single classification tree. Again, this isn't surprising as random forests and boosting were both conceived in order to improve on the single decision tree by combining many trees, thereby decreasing variance. Also of note is that every single classification method identified `perimeter_worst` and `concave.points_worst` as the most important predictors in determining whether a tumor is benign or malignant.

V. Citations

Combescot, Roland (2022). Superconductivity. Cambridge University Press. pp. 1–2. ISBN 9781108428415.

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani. An Introduction to Statistical Learning: with Applications in R. New York :Springer, 2013.

Hamidieh, Kam, A data-driven statistical model for predicting the critical temperature of a superconductor, Computational Materials Science, Volume 154, November 2018, Pages 346-354, [https://doi.org/10.1016/j.commatsci.2018.07.052]

W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction for breast tumor diagnosis. (1995). UCI Machine Learning Repository [https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29]. Irvine, CA: University of California, School of Information and Computer Science.