Challenge B

Sharanya Pillai & Jonas Gathen

26 November 2017

The Github repository for this analysis can be found here. The associated SSH-key is: git@github.com:jgathen/R_prog_ChallengeB.git The associated HTTPS-key is: https://github.com/jgathen/R_prog_ChallengeB.git

Task 1B: Fitting a random forest to predict housing prices

Step 1

We are choosing a random forest. The following description and intuition of the method is mostly based on Hastie, Tibshirani & Friedman (2008) and James, Witten, Hastie & Tibshirani (2014): Random forests are based on tree-based methods. Tree-based methods work through iteratively segmenting the space of possible predictions based on certain splitting rules. The collection of these splitting rules can then be summarized in a tree-form, hence the name of this type of methods. Random forests represent a way in which to combine the predictions of multiple trees to form one combined prediction. It is especially linked to the method of bagging. The essential idea in bagging is to reduce variance by averaging many noisy but approximately unbiased models. Trees are ideal candidates for bagging, since they can capture complex interaction structures in the data, and if grown sufficiently deep, have relatively low bias. Since trees are notoriously noisy, they benefit greatly from the averaging. Each tree generated in bagging is identically distributed, but not necessarily independent. This has the unfortunate consequence that even for many trees, the variance of the average of trees will face a lower bound that is given by the positive pairwise correlation of trees.

The idea in random forests is to improve the variance reduction of bagging by reducing the correlation between the trees, without increasing the variance too much. This is achieved by the following general algorithm:

- Choose the size of your forest (i.e. how many trees you want to predict).
- Then for each tree, draw a bootstrap sample from the training data.
- With this sample, select a number m of variables at random from the total number of variables p.
- Pick the best variable among the number of variables that were randomly drawn (based on splitting criterium) and split accordingly.
- Again, randomly draw a prespecified number of variables from the total, pick the best variable and split. Repeat until a specified minimum node size is reached.
- Save the resulting tree and repeat procedure to get the remaining trees.
- The random forest predictor is given by the average of the trees.

The correlation between trees can be controlled by m. The smaller m, the smaller the correlation will be. However, with large p (the total number of variables) and many non-relevant variables, a small m can lead to weak performance. This is not likely to be a problem in this case here, because p is not very large and we actually have a large number of important variables. As a practical advice, for regression, the default value for m is p/3 and the minimum node size is five; this should be fine-tuned however and can vary significantly from one problem to another.

Step 2: Train the chosen technique on the training data

We read the revised training data from the last challenge and train the model using the above-outlined algorithm for a random forest. We stick with the recommended default of 500 trees per forest and a terminal node size of 5. We also use the default value for m as p/3. Usually, we would call the random forest algorithm from within the caret-package to fine-tune the parameter m by using 5- or 10-fold cross-validation, using only the training set (because we don't have the necessary observations for using the test dataset). This takes too long, though; and the exercise is only meant to illustrate anyway.

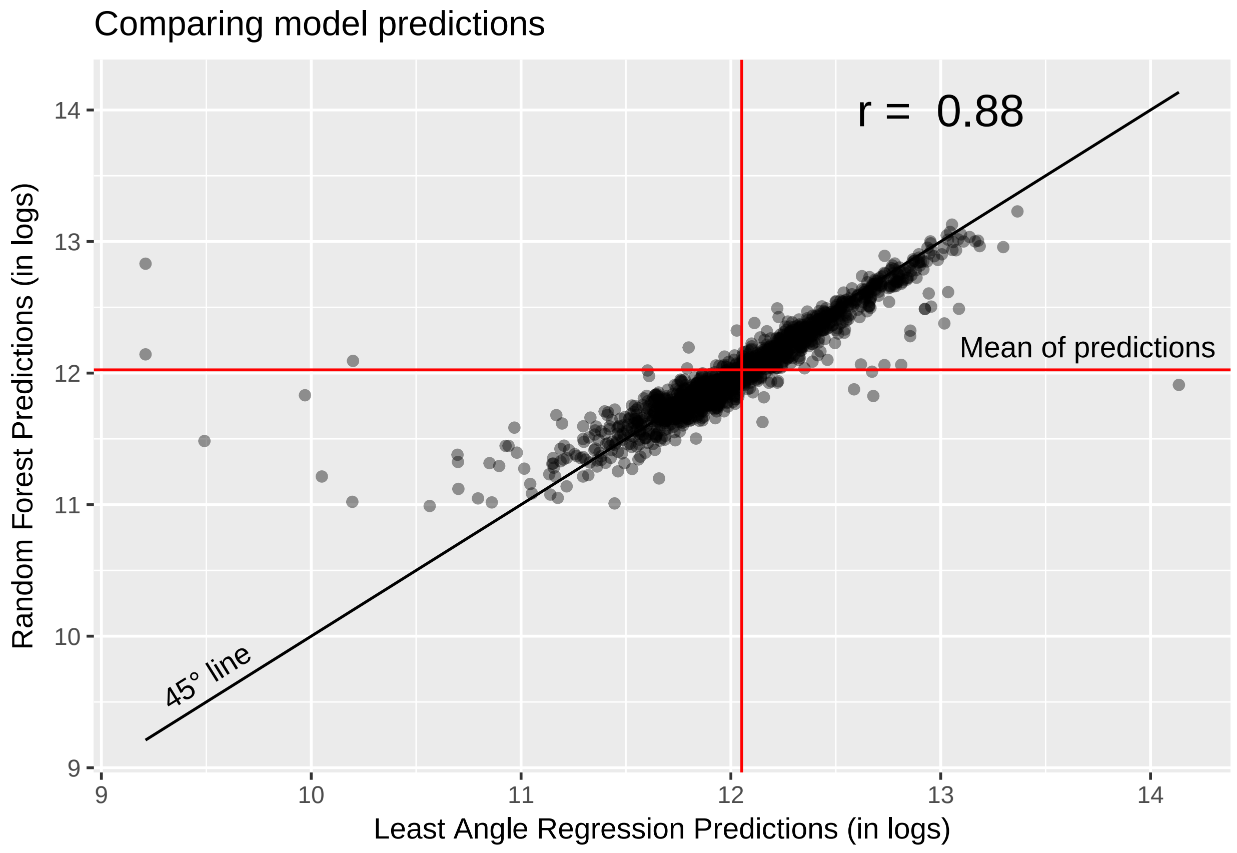
```
training <- read.csv(file = "training_final_ChallengeA.csv", header = T)
random_forest_model1 <- randomForest(data=training, SalePrice~.-Id, ntree = 500, nodesize = 5)
```

Step 3: Make predictions on the test data, and compare them to the predictions of a linear regression of your choice.

We read the revised test dataset from the previous challenge. We use our trained model from above to make predictions on the test data. We then compare these predictions to the predictions obtained in the previous challenge, where we fitted a final model with Least Angle Regression.

```
test <- read.csv(file = "test_final_ChallengeA.csv", header = TRUE)
rf_predictions <- predict(random_forest_model1, test)
lar_predictions <- read.csv(file = "final_predictions_ChallengeA.csv", header = TRUE)
compare_predictions <- cbind(lar_predictions,rf_predictions)
```

We can now compare the predictions (without being able to look at the true values). This is best illustrated by plotting the predictions against each other. The figure below shows that they are fairly close (r = 0.88). Main differences arise for extreme observations for which both models predict lower or higher prices on average. In general, the Least Angle Regression predicts more extreme values; for prices below average, the Least Angle regression model predicts much lower prices, for prices above average, the Least Angle regression predicts higher prices.



Task 2B - Overfitting in Machine Learning (continued) - 1 point for each step

We can briefly create the data we need for this exercise.

```
nsims <- 150 # Number of simulations
e <- rnorm(n = nsims, mean = 0, sd = 1) # Draw 150 errors from a normal distribution
x <- rnorm(n = nsims, mean = 0, sd = 1) # Draw 150 x obs. from a normal distribution
y <- x^3+e # generate y following (T)
df <- data.frame(y,x)

df$ID <- c(1:150)
training2 <- df[df$ID %in% sample(df$ID, size = 120, replace = F), ] # Get training set of size 120
test2 <- df[!(df$ID %in% training2$ID), ] # Get remaining test dataset
df$training <- (df$ID %in% training2$ID) # Create variable specifying whether obs. is in test or traini
```

Step 1: Estimate a low-flexibility local linear model on the training data

```
ll.fit.lowflex <- npreg(training2, formula = y ~ x, method = "ll", bws = 0.5)
```

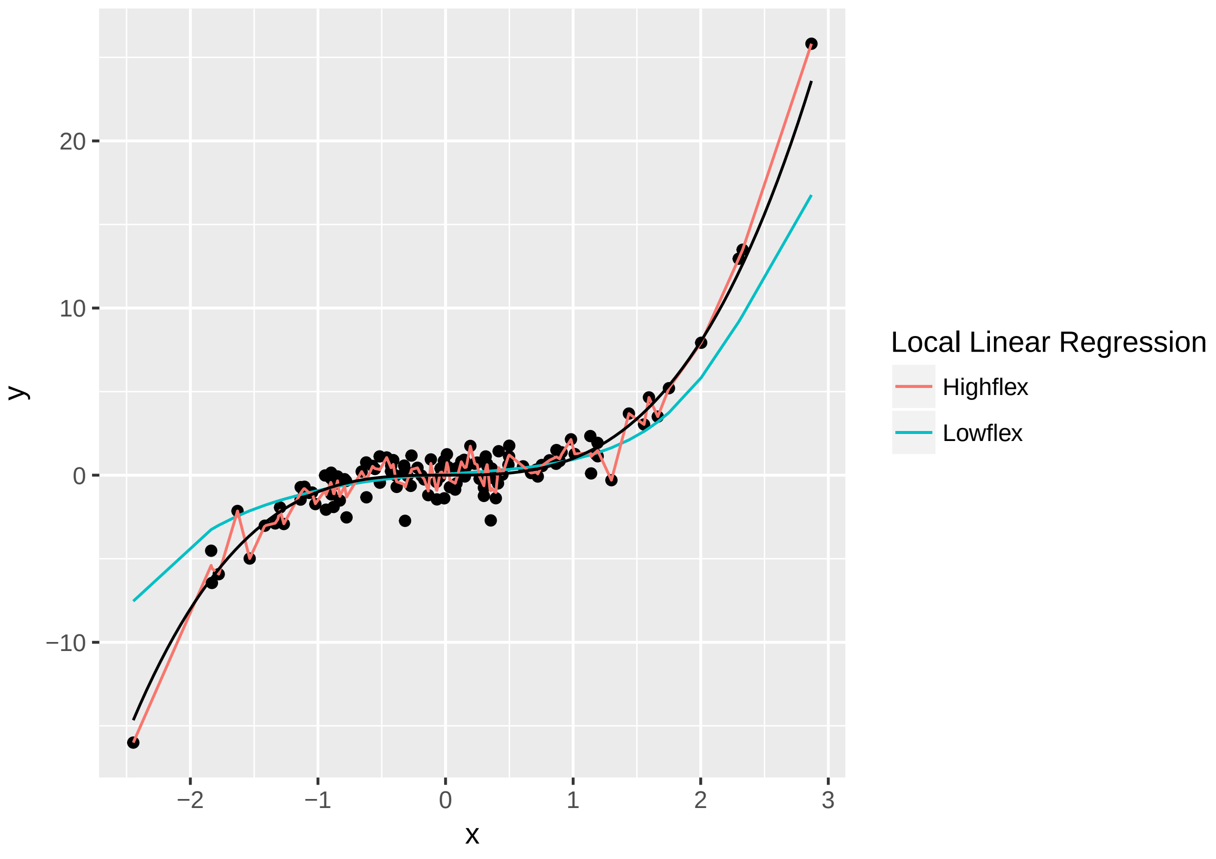
Step 2: Estimate a high-flexibility local linear model on the training data.

```
ll.fit.highflex <- npreg(training2, formula = y ~ x, method = "ll", bws = 0.01)
```

Step 3: Plot the scatterplot of x-y, along with the predictions of ll.fit.lowflex and ll.fit.highflex, on only the training data.

```
# Get estimates of both models for training2 data
lowflex_estimates <- data.frame(y_estimates_lowflex = ll.fit.lowflex$mean, y = training2$y, x = ll.fit.l
highflex_estimates <- data.frame(y_estimates_highflex = ll.fit.highflex$mean, y = training2$y, x = ll.fi
combined_estimates <- merge(lowflex_estimates, highflex_estimates)

ggplot(data = combined_estimates) + geom_point(aes(x = x, y = y)) +
  geom_line(aes(x = x, y = y_estimates_lowflex, color = "red")) +
  geom_line(aes(x = x, y = y_estimates_highflex, color = "darkblue")) +
  stat_function(fun = function(x) x^3) +
  scale_color_discrete(name = "Local Linear Regression", labels = c("Highflex", "Lowflex"))
```



Step 4 - Between the two models, which predictions are more variable? Which predictions have the least bias?

In machine learning, we look at models from the perspective of the bias-variance trade-off, where bias is

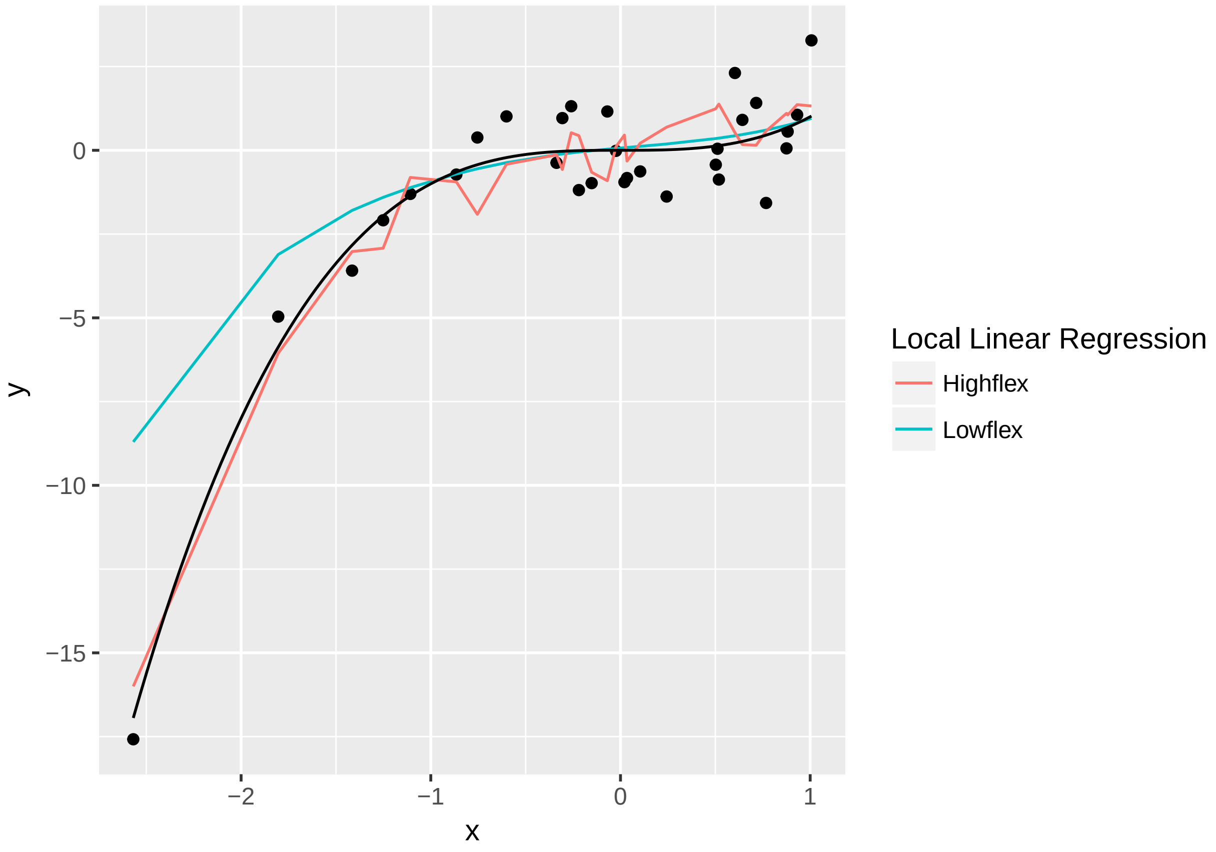
the variance of the learning method,or, intuitively, how much the learning method will move around its mean

Judging only by looking at the predictions on the training dataset, the high-flexibility local linear model outperforms the low-flexibility-model in both bias and variance. The high-flexbility model has a low bias This is expected, because we only face a bias-variance trade-off when applying the model to the test data. The highfle

Step 5 - Plot the scatterplot of x-y, along with the predictions of ll.fit.lowflex and ll.fit.highflex now using the test data. Which predictions are more variable? What happened to the bias of the least biased model?

```r
# Get predictions of both models for test2 data
lowflex_predictions <- predict(ll.fit.lowflex, newdata = test2)
highflex_predictions <- predict(ll.fit.highflex, newdata = test2)
combined_predictions <- cbind(test2, lowflex_predictions, highflex_predictions)

ggplot(data = combined_predictions) + geom_point(aes(x = x, y = y)) +
  geom_line(aes(x = x, y = lowflex_predictions, color = "red")) + # Plot predictions from lowflex
  geom_line(aes(x = x, y = highflex_predictions, color = "darkblue")) + # Plot predictions from highfle
  stat_function(fun = function(x) x^3) +
  scale_color_discrete(name = "Local Linear Regression", labels = c("Highflex", "Lowflex"))
```



Compare bias and variance.

Step 6 - Create a vector of bandwidth going from 0.01 to 0.5 with a step of 0.001

```
bandwidth_vector <- seq(0.01,0.5,0.001)
```

Step 7 - Estimate a local linear model y ~ x on the training data with each bandwidth.

We can either do this via looping or in vectorized form. We don't expect there to be any differences in computation time however, because we are running a different regression for each bandwidth. If we would apply the exact same function to each of the vector's elements, then vectorization could save a lot of time.

```
run_ll <- function(bandwidth){
  npreg(training2, formula = y ~ x, method = "ll", bws = bandwidth)
}
system.time(ll_models_apply <- lapply(X = bandwidth_vector, FUN = run_ll))
```

```
##    user  system elapsed
##   3.938   0.133   4.209
```

```
ll_models_loop <- list()
system.time(for(i in bandwidth_vector){
  ll.fit <- npreg(training2, formula = y ~ x, method = "ll", bws = i)
  ll_models_loop[[length(ll_models_loop)+1]] <- ll.fit
})
```

```
##    user  system elapsed 
##   3.622   0.082   3.874
```

Step 8 - Compute for each bandwidth the MSE on the training data.

In the next step, we can just extract the already computed MSE from our model output. Again, vectorizing or looping take almost the same amount of time.

```
system.time(MSE_training <- sapply(c(1:length(bandwidth_vector)), FUN = function(i) ll_models_apply[[i]]
```

```
##    user  system elapsed 
##   0.002   0.000   0.003
```

```
MSE_training <- c()
system.time(for(i in 1:length(bandwidth_vector)){
 MSE_training[i] <- ll_models_loop[[i]]$MSE
})
```

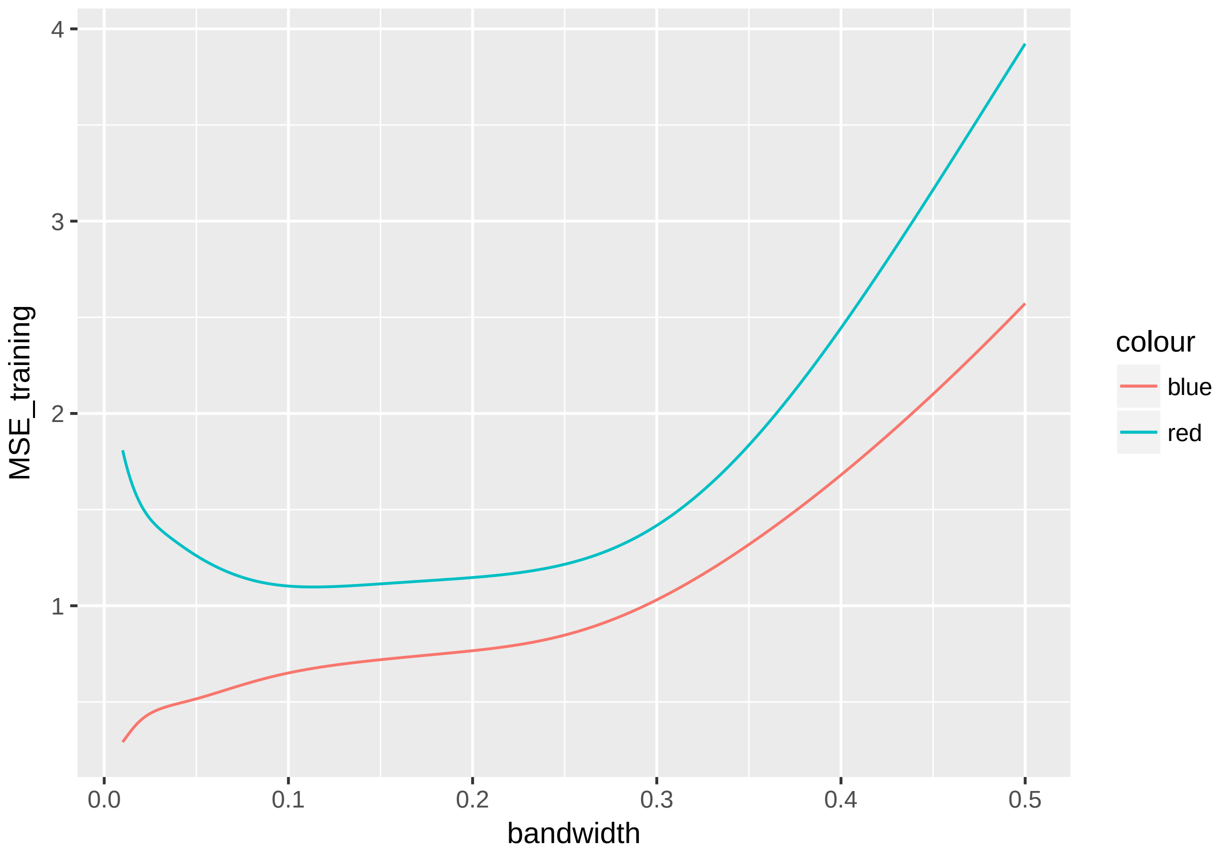
```
##    user  system elapsed 
##   0.002   0.000   0.003
```

Step 9 - Compute for each bandwidth the MSE on the test data.

```
MSE_test <- c()
for(i in 1:length(bandwidth_vector)){
  ll_model_predictions <- predict(ll_models_loop[[i]], newdata = test2) # Get prediction for given band
  MSE_test[i] <- mean((test2$y-ll_model_predictions)^2) # Compute mean squared error for given bandwidt
}
```

Step 10 - Draw on the same plot how the MSE on training data, and test data, change when the bandwidth increases. Conclude.

```
MSE <- data.frame(bandwidth = bandwidth_vector, MSE_training, MSE_test) # Combine in one dataset
ggplot(data = MSE) +
  geom_line(aes(x = bandwidth, y = MSE_training, color = "blue")) +  # MSE from training
  geom_line(aes(x = bandwidth, y = MSE_test, color = "red"))         # MSE from test
```



Task 3B: Privacy regulation compliance in France