HarvardX PH125.9xData Science: Capstone: Severstal Steel Defects

Yannick Hermans

15-11-2020

Introduction

In 2019 Severstal launched a competition on kaggle (https://www.kaggle.com/c/severstal-steel-defect-detection/overview/description) where participants were asked to develop algorithms which could swiftly and accurately detect and classify certain defects in manufactured steel samples. The winners were promised a prize worth $40,000 with runner ups receiving smaller cash prizes. The challenge was to detect four types of defects in a series of steel sample images. The data consisted of a set of training images, a set of test images, and a csv file containing three variables: Image identifier, Defect identifier and a string with numbers which identify the pixels in the image where the defect was present.

In this second capstone project, which forms the conclusion to the HarvardX PH125.9xData Science course, the goal is to classify the steel defect type using the information in the Severstal data set. New features from the data will be created and will be used by common machine learning algorithms for the classification of the defect type using the accuracy as benchmark. This project will represent the knowledge I gained throughout the edX data science course on machine learning algorithms, data wranlging, data exploration and data visualization in r.

Data wrangling and exploration

First the libraries needed for the data wrangling, data exploration and machine learning development are installed and loaded

```
if(!require(tidyverse)) install.packages("tidyverse",
  repos = "http://cran.us.r-project.org")
if(!require(imager)) install.packages("imager",
  repos = "http://cran.us.r-project.org")
if(!require(stringr)) install.packages("stringr",
  repos = "http://cran.us.r-project.org")
if(!require(knitr)) install.packages("knitr",
  repos = "http://cran.us.r-project.org")
if(!require(gridExtra)) install.packages("gridExtra",
  repos = "http://cran.us.r-project.org")
if(!require(purrr)) install.packages("purrr",
  repos = "http://cran.us.r-project.org")
if(!require(caret)) install.packages("caret",
  repos = "http://cran.us.r-project.org")

library(tidyverse)
library(imager)
library(stringr)
library(knitr)
library(gridExtra)
library(purrr)
library(caret)
```

Next the data were downloaded from https://www.kaggle.com/c/severstal-steel-defect-detection/data and saved in a subdirectory (of the project directory) called Data. Three items were downloaded: train.csv (file), train_images (directory), test_images (directory). The train_images and test_images directories contain images for training and testing machine algorithms, respectively. The image files were not directly loaded into the workspace due to their sheer file size. The train.csv file contains the defect information on the train images.

The train.csv file is loaded into the project:

```
train <- readr::read_csv("./Data/train.csv")
```

The train data set can be visualized:

```
## # A tibble: 6 x 3
##   ImageId      ClassId EncodedPixels
##   <chr>          <dbl> <chr>
## 1 0002cc93b.j~       1 29102 12 29346 24 29602 24 29858 24 30114 24 30370 24 30~
## 2 0007a71bf.j~       3 18661 28 18863 82 19091 110 19347 110 19603 110 19859 11~
## 3 000a4bcdd.j~       1 37607 3 37858 8 38108 14 38359 20 38610 25 38863 28 3911~
## 4 000f6bf48.j~       4 131973 1 132228 4 132483 6 132738 8 132993 11 133248 13 ~
## 5 0014fce06.j~       3 229501 11 229741 33 229981 55 230221 77 230468 92 230623~
## 6 0025bde0c.j~       3 8458 14 8707 35 8963 48 9219 71 9475 88 9731 88 9987 89 ~
```

It can be seen that the data is tidy with rows being observations and columns being variables. Each observation corresponds to a particular defect spanning a certain range of pixels. The train data set contains three variables: 1) ImageId: a unique character string corresponding to a certain image in the train_images directory 2) ClassId: a numeric Id which identifies the type of defect 3) EncodedPixels: a character string which identifies the parts of the image which contain defects of the type specified in the ClassId. The string contains a repeating pattern of two kinds of numbers, the first is the pixel where the defect starts and the second is the number of pixels which contains the defect, counting from the start pixel moving right along the particular row of pixels.

Data exploratory analysis

Plotting image with defects

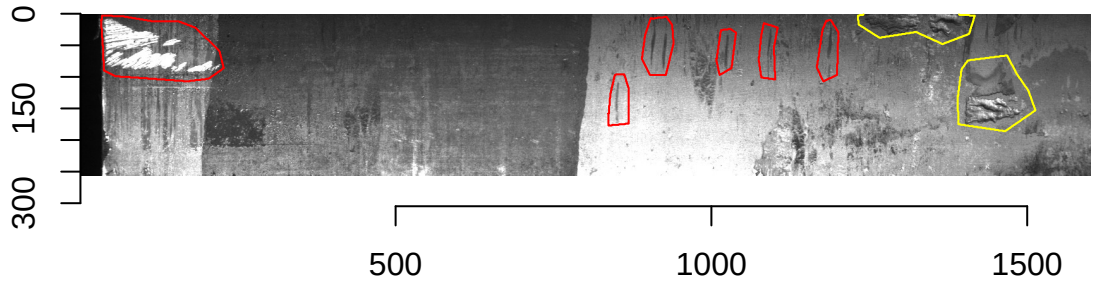
To plot the steel images with the different defects indicated in color I have adapted the code provided by Ismail Müller on https://www.kaggle.com/mullerismail/steel-defect-understand-plot-with-r-novice:

```
split_pixels_single_class <- function(EncodedPixels){
  str_split(EncodedPixels, "[:space:]", simplify = TRUE) %>%
    # splits the EncodedPixel observation using :space: character as a pattern
    as.numeric() %>%
    # the valus are still characters, so we transform them into numerical values
    matrix(ncol = 2, byrow = TRUE) %>%
    # create a matrix of 2 columns with the values
    as.data.frame() %>% # transform the matrix into a data.frame
    transmute(pixel_start = V1, run_length = V2,
              pixel_end = pixel_start + run_length - 1)
  # create a variable pixel_end
}
split_pixels_multi_class <- function(image_name){
  train %>% filter(ImageId == image_name, !is.na(EncodedPixels)) %>%
    split( .$ClassId ) %>%
    map( ~ split_pixels_single_class(.$EncodedPixels) %>%
           pmap(~ seq(..1, ..3, by=1)) %>% unlist )
}
```

```
plot_image <- function(image_name){
  # find the image
  path_to_image  <- list.files(path = "./Data/train_images",
                               pattern = image_name,recursive = TRUE, full.names=TRUE)
  if( length(path_to_image) == 0 )
    { stop(str_glue("Image {image_name} wasn't found in the images")) }

  # load the image
  image_to_plot <- path_to_image %>% imager::load.image() %>% imager::grayscale()
  # Create the masks for each defect class
  height <- dim(image_to_plot)[2]
  width <- dim(image_to_plot)[1]

  masks <- split_pixels_multi_class(image_name) %>%
    map(~ {
      m <- matrix(0, nrow=height, ncol=width)
      m[.x] <- 1
      t(m) %>% as.cimg() %>% as.pixset()
    })

  # plot the image and the defects
  plot(image_to_plot)
  for(i in 1:length(masks)) {
    highlight(masks[[i]],
              col = c("blue","green","red","yellow")[as.numeric(names(masks)[i])] )
  }
}
```

Indeed, using the function plot_image the defects in the steel samples can be visualized using the ImageId as argument.

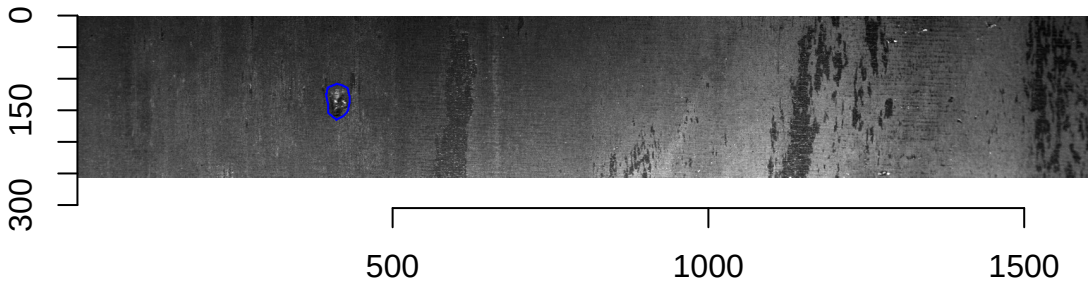
```
plot_image("0025bde0c.jpg")
```



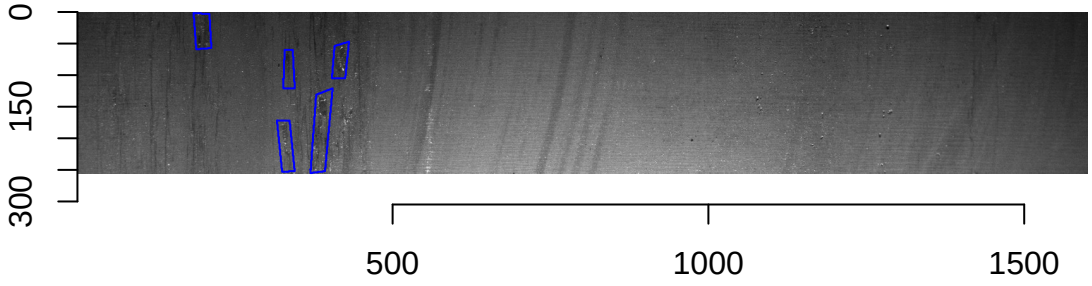
Types of defect

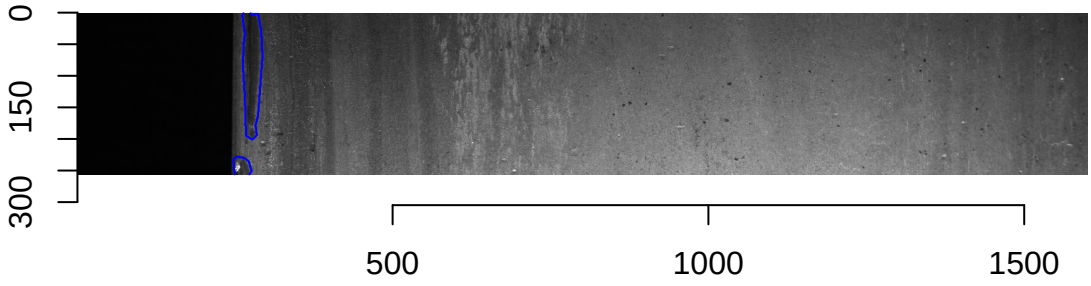
To have an idea how the different defects look like, a set of five images per defect type is plotted underneath. The images were randomly selected for each type according to:

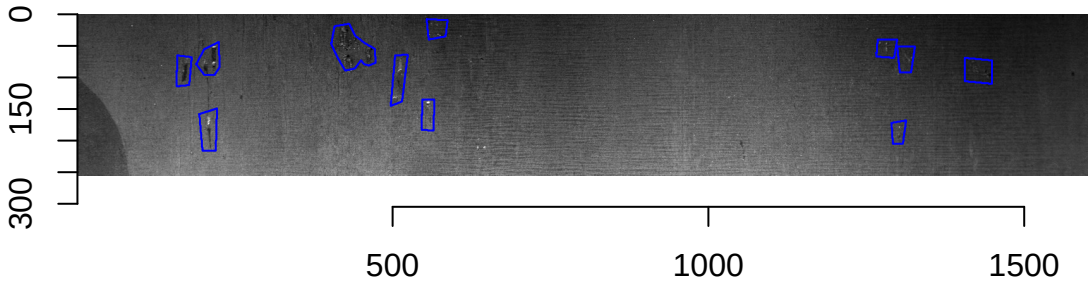
```
train_type1 <- train %>% filter(ClassId == 1)
t <- as.character(train_type1[sample(1:nrow(train_type1),1),1])
plot_image(t)
```

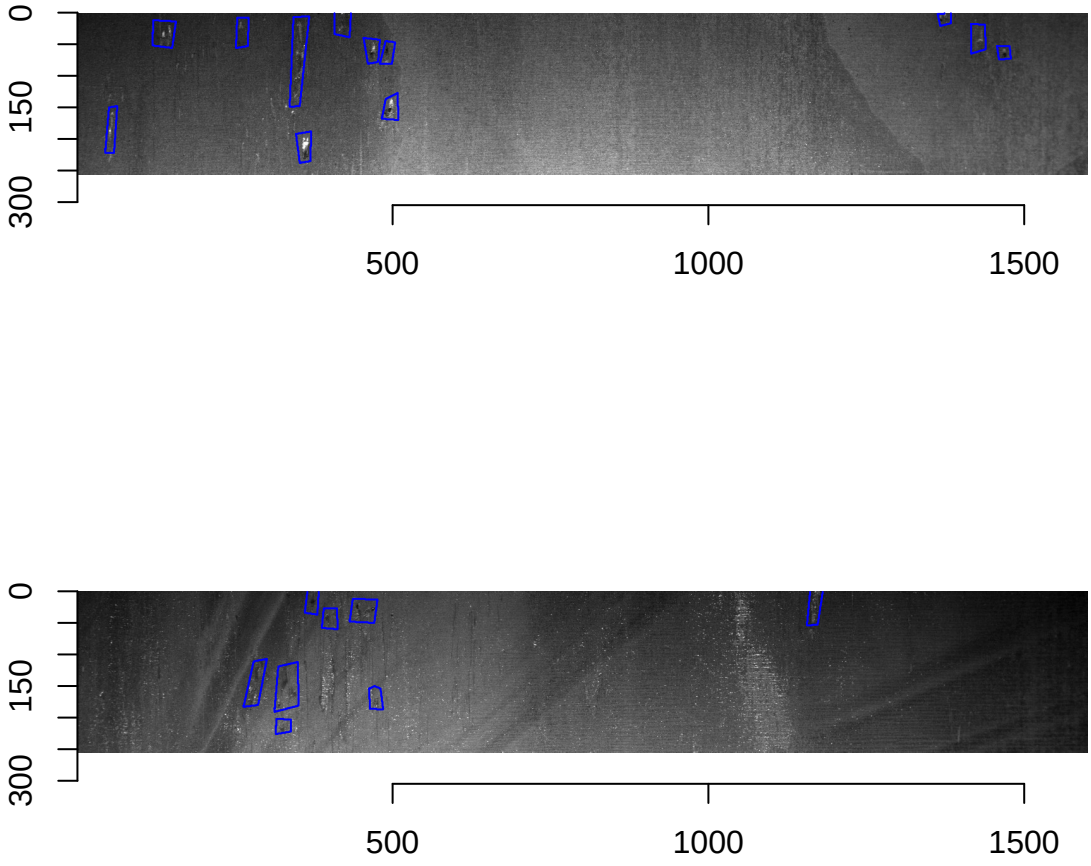


TYPE 1



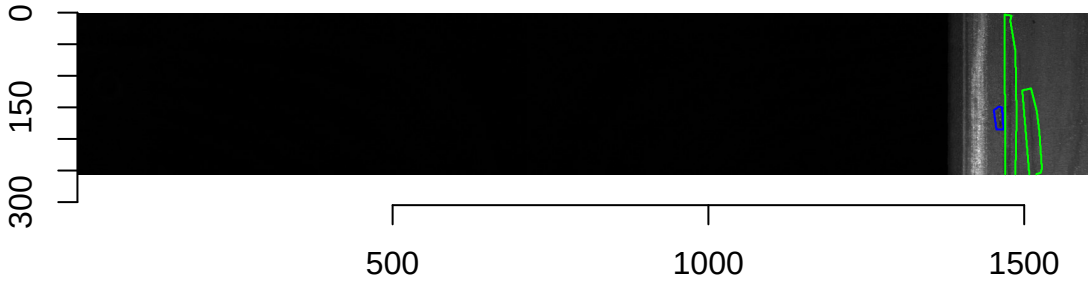

0
150
300
500
1000
1500

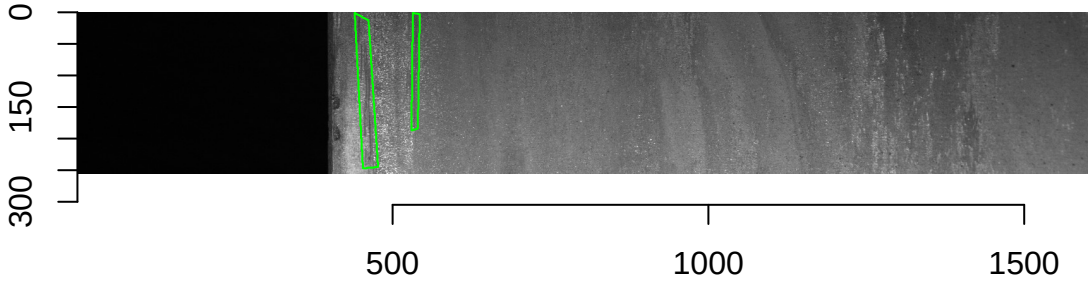

0
150
300
500
1000
1500

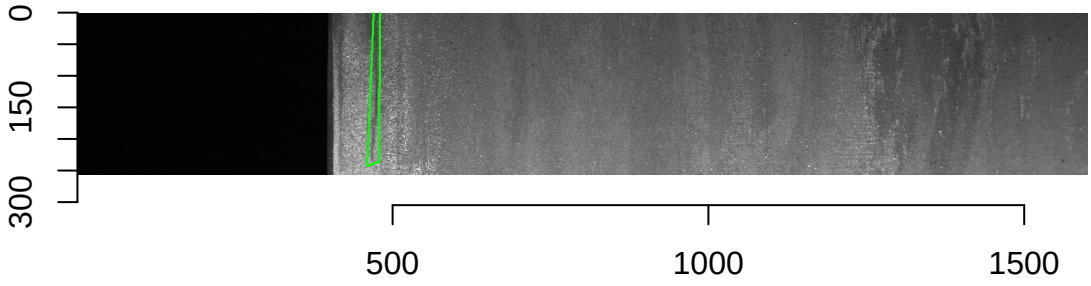


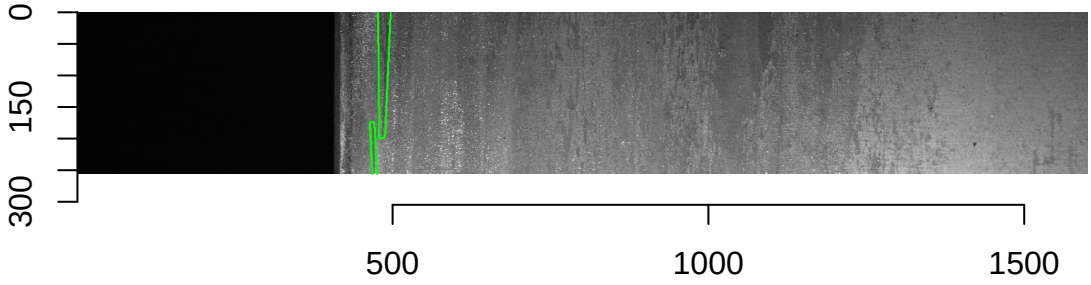
In these sample images this type of defect appears to be a point defect as it spans only a small cluster of pixels for most defects. Visually, the pixels spanning the defect contrast stongly with the environment being more black or white. Most defects contain a few black and a few white pixels.

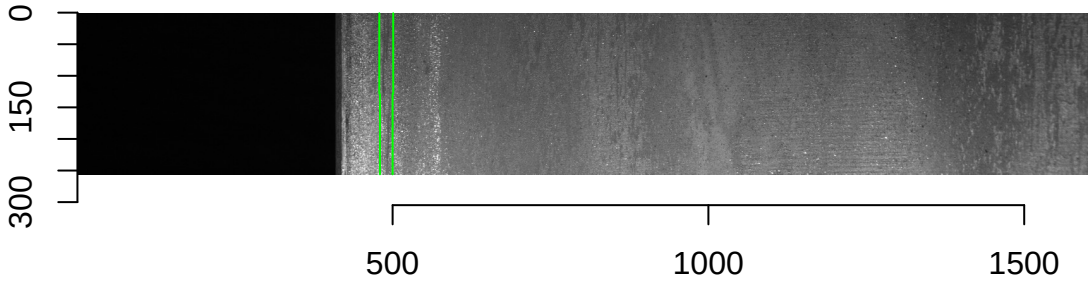
TYPE 2


0
150
300
500
1000
1500


0
150
300
500
1000
1500

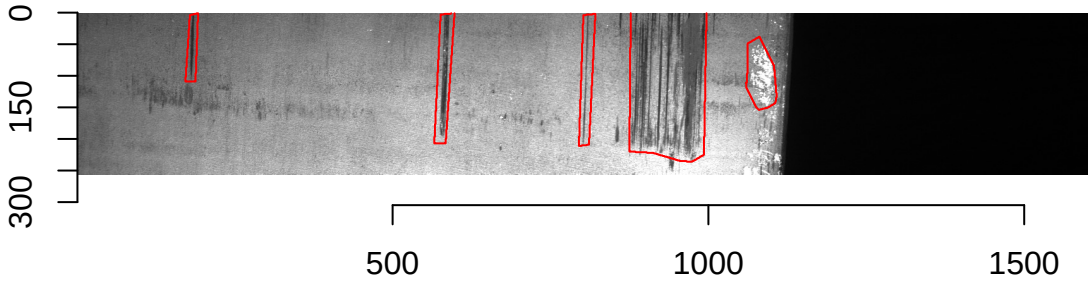

0
150
300
500
1000
1500

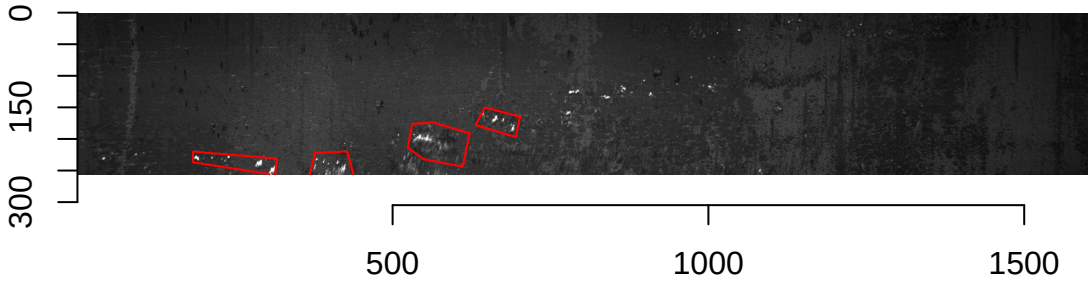

0
150
300
500
1000
1500

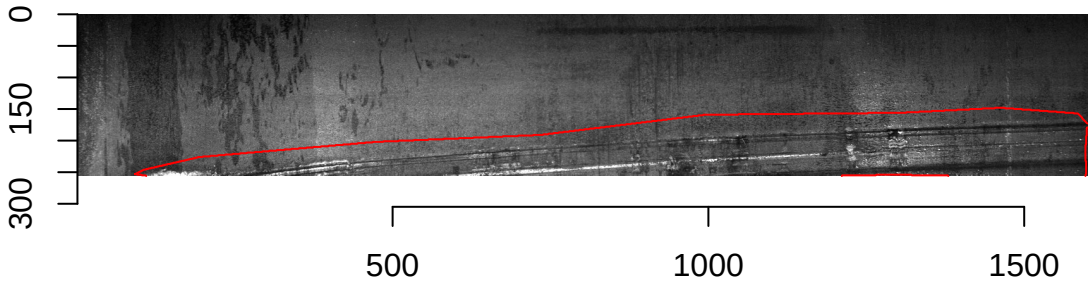


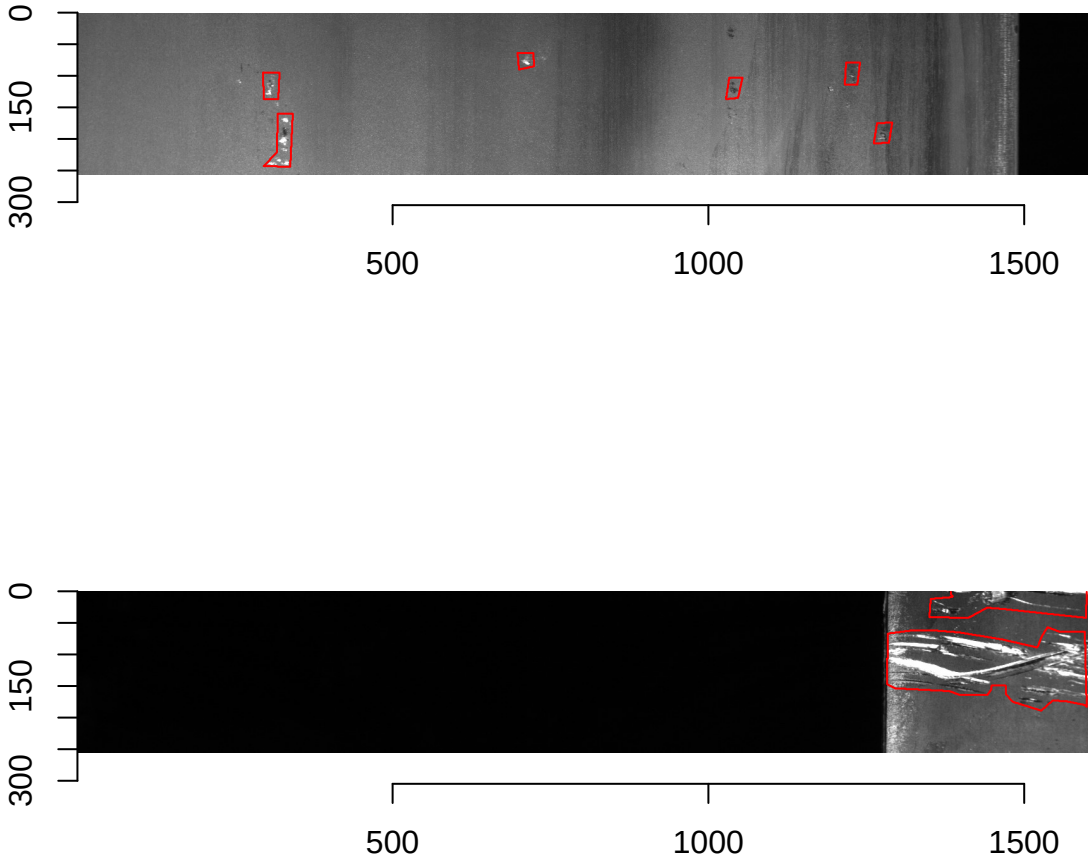
The defects in the sample images look very similar to each other. The defects align vertically and there is a slight contrast between the pixels spanning the defect and the environment. The defect pixels are a bit darker. The defect spans almost the entire height of the image.

TYPE 3



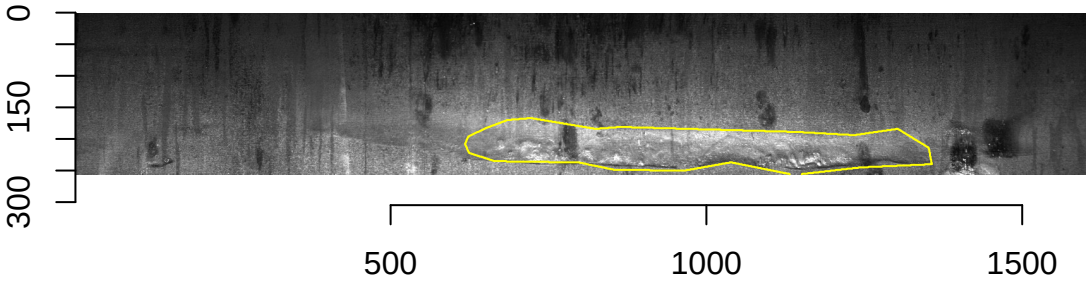

0
150
300
500
1000
1500

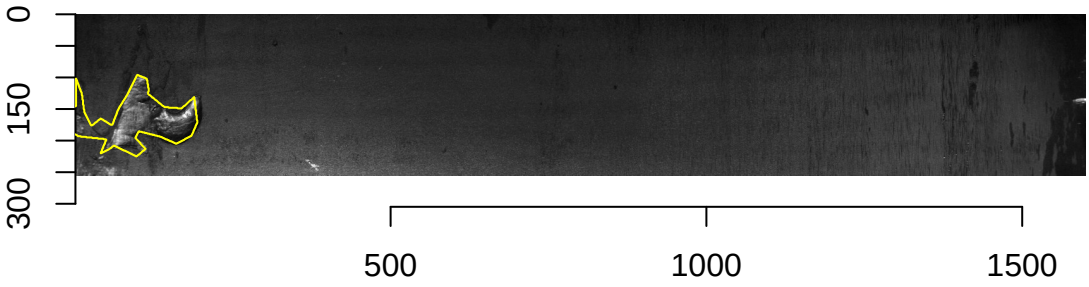

0
150
300
500
1000
1500

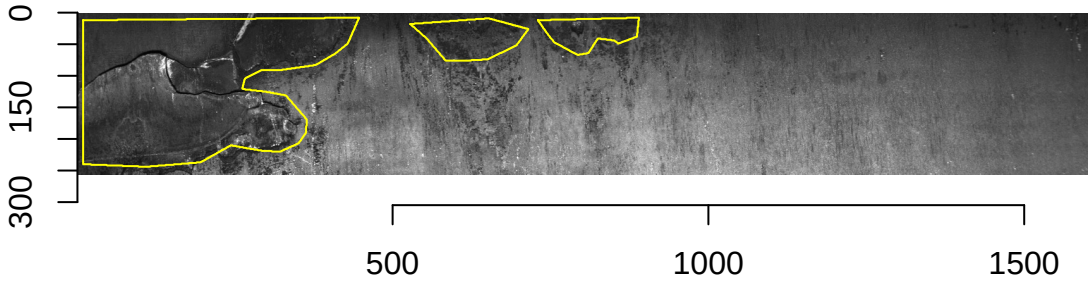


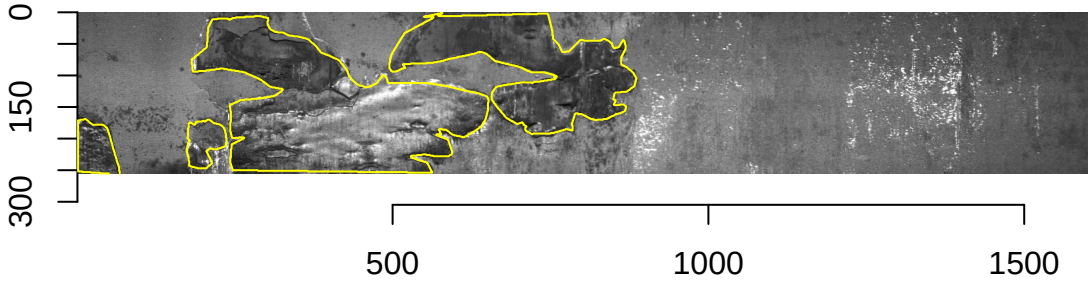
The type 3 defect seems to be quite varied, since it sometimes spans a small cluster of pixels and sometimes a large cluster. The defects appear to be scratchlike in nature. The pixels spanning the defect also contrast more strongly with the environment than defect 2. The pixels can be darker or lighter than the environment.

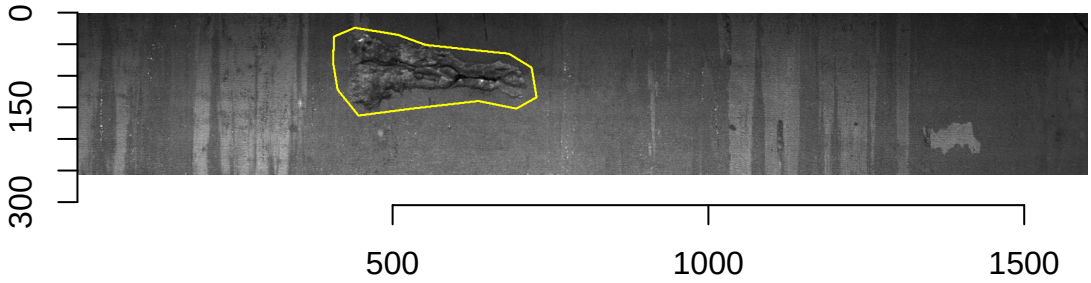
TYPE 4


0
150
300
500
1000
1500


0
150
300
500
1000
1500

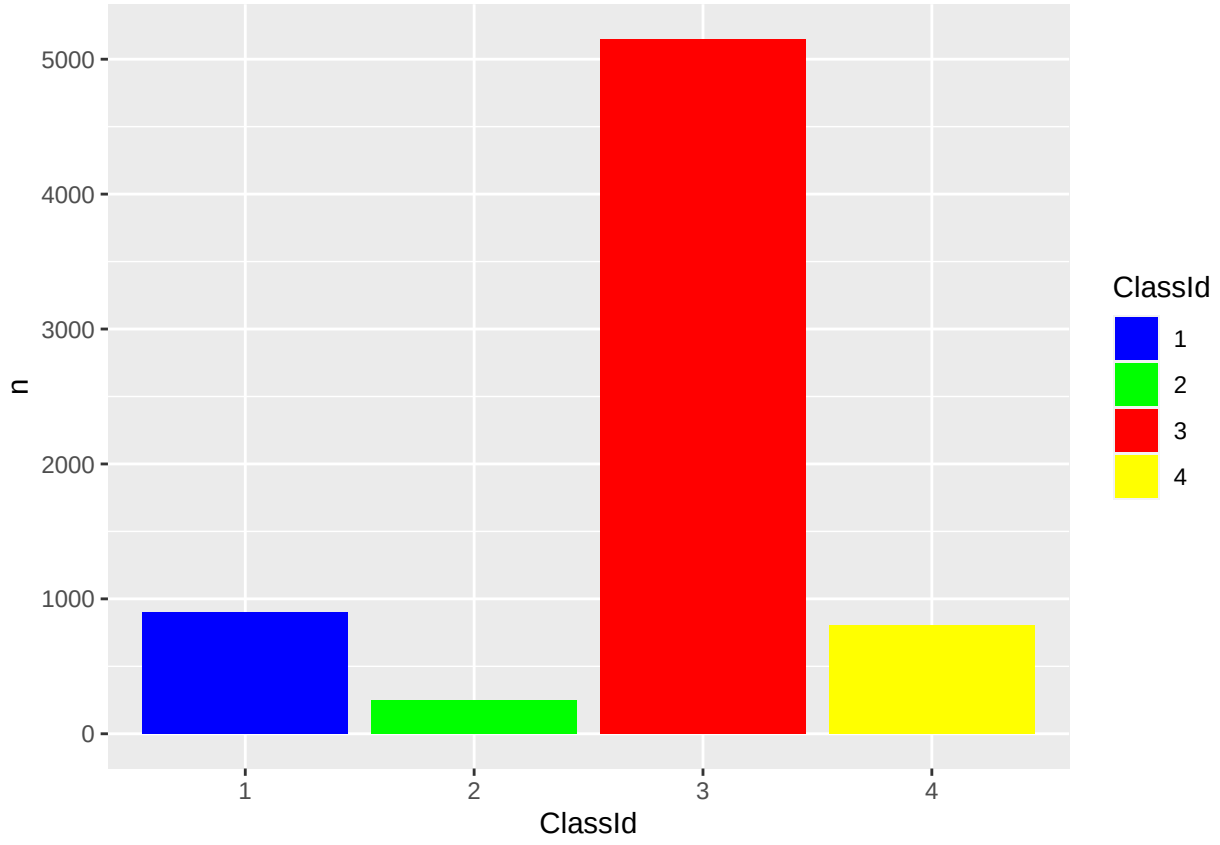

0
150
300
500
1000
1500


0
150
300
500
1000
1500



These type of defects appear to be quite large in general, spanning large clusters of pixels. Within these defects there is a strong variability in the greytone of the pixels, much more so than the environment and within other defects.

Height, width and size of defects



The train data set contains considerably more type 3 defects than any of the other defects. This could be due to this type of defect being more prevalent during manufacturing or because the company wanted to include

more images of type 3, because the company finds the detection of type 3 defects more important. More images could also be included, because of the strong variability of the type 3 defects.

From the random sampling of the images it could be seen that the width, height and size of the defects may be used to identify the defect. Using the code underneath the size, height and width of the defects are extracted from the EncodedPixels variable.

```
add_width_height_pixels <- function(train_set){
  train_set_encoded_pixels <- as.list(train_set$EncodedPixels)
  defect_width_height_pixels <- function(i){
    columnlist <- seq(from = 1, to = 4096000, by = 256)
    # list containing the first pixel of every column
    i <- i %>% str_split(., "[:space:]") %>%
      # splits the EncodedPixel observation using :space: character as a pattern
      unlist() %>% # unlists/as.vector the result of str_split
      as.numeric() %>%
      # the valus are still characters, so we transform them into numerical values
      matrix(ncol = 2, byrow = TRUE) %>%  # create a matrix of 2 columns with the values
      as.data.frame() %>% # transform the matrix into a data.frame
      rename(pixel_start = V1, run_length = V2)
    widthlist <- function(j) {ifelse(any(j:(j+255) %in% i$pixel_start),1,0)}
    width <- sum(sapply(columnlist, widthlist)) # function needed to determine the width
    i %>% summarize(height = ifelse(max(run_length) > 256, 256, max(run_length)),
                    width = width , total_pixels = sum(run_length)) %>%
      unlist()
  }
  width_height_pixels_per_defect <-
    as.data.frame(t(as.matrix(sapply(train_set_encoded_pixels, defect_width_height_pixels))))
  width_height_pixels_per_defect_image <-
    cbind(width_height_pixels_per_defect, ImageId = train_set$ImageId, ClassId = train_set$ClassId)
  right_join(train_set,width_height_pixels_per_defect_image)
}
train2 <- add_width_height_pixels(train)
```

The altered train data set is named as train2 and looks now like this:

```
## # A tibble: 7,095 x 6
##    ImageId   ClassId EncodedPixels                       height width total_pixels
##    <chr>       <dbl> <chr>                               <dbl> <dbl>        <dbl>
##  1 0002cc93~       1 29102 12 29346 24 29602 24 29858~    105    95         4396
##  2 0007a71b~       3 18661 28 18863 82 19091 110 1934~    251    37         6897
##  3 000a4bcd~       1 37607 3 37858 8 38108 14 38359 2~     50   223         8319
##  4 000f6bf4~       4 131973 1 132228 4 132483 6 13273~    202   617        69357
##  5 0014fce0~       3 229501 11 229741 33 229981 55 23~    186    34         4851
##  6 0025bde0~       3 8458 14 8707 35 8963 48 9219 71 ~     97   377        28506
##  7 0025bde0~       4 315139 8 315395 15 315651 16 315~    121   282        17541
##  8 002af848~       4 290800 6 291055 13 291311 15 291~     58   163         5001
##  9 002fc4e1~       1 146021 3 146275 10 146529 40 146~     78    26         1547
## 10 002fc4e1~       2 145658 7 145901 20 146144 33 146~     93    17         1209
## # ... with 7,085 more rows
```

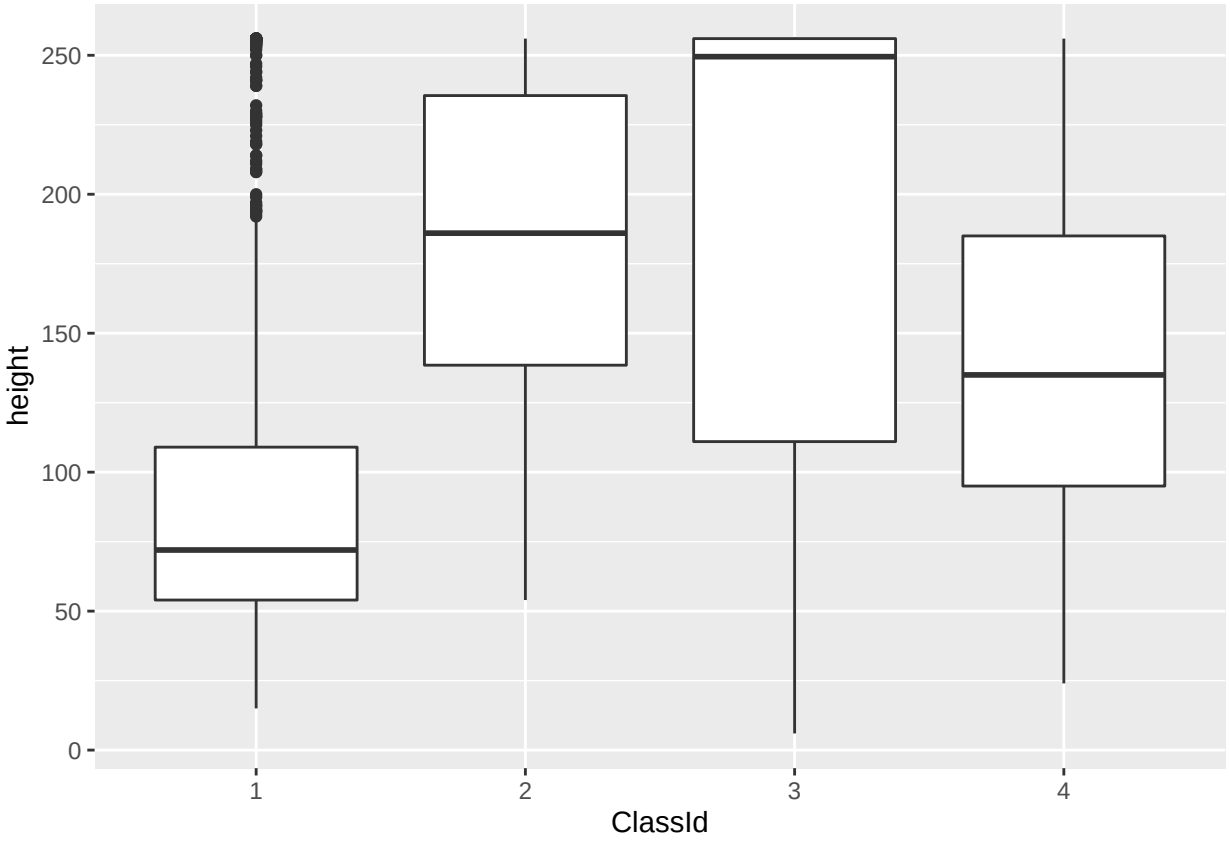
It's however important to note that the EncodedPixels variable contains multiple defects, which has been neglected for in the code above. In the calculation of width, height and size the EncodedPixels variable is considered to represent one defect. The height then amounts to the height of the tallest defect, the width amounts to the number of columns which contain pixels encoding for a defect and total_pixels are all the pixels encoding for that defect. Separating the defects would be a time consuming effort since the

EncodedPixels variable should then actually be transformed in a list of matrices so that clusters of pixels can be recognized.

Anyway, the new variables may already yield some information to distinguish the defects from one another. The different classes of defects are compared with one another using boxplots of the new variables. In addition, some outliers are visualized to obtain more information of the morphology of these outliers.

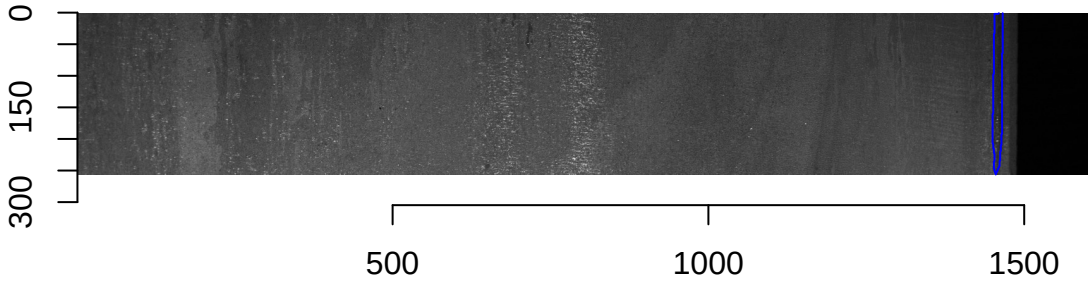
The first series of boxplots show the distribution of heights for each type of defect.

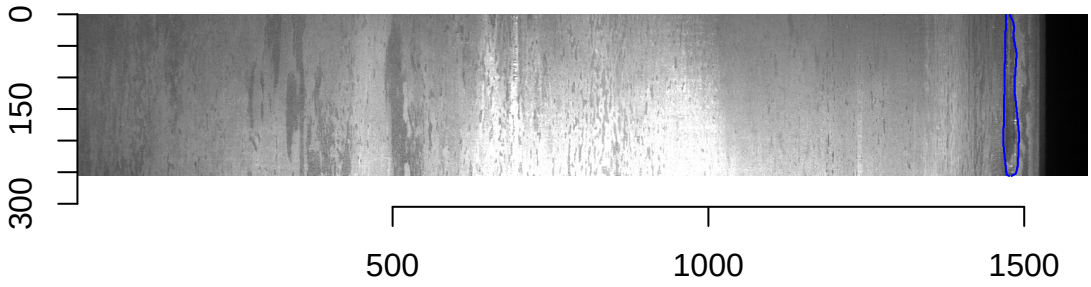
HEIGHT

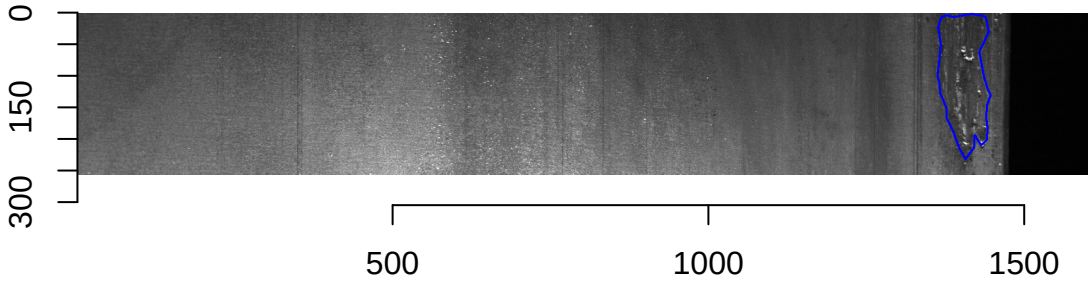


The distribution of the height is quite different for the different kind of defects. The first type of defect has in average the lowest height and the lowest variability, while type 3 has the strongest variation in height and the highest average height.

Although the average of defect 1 is quite low, it has many outliers with increased height. Let's look at three examples.

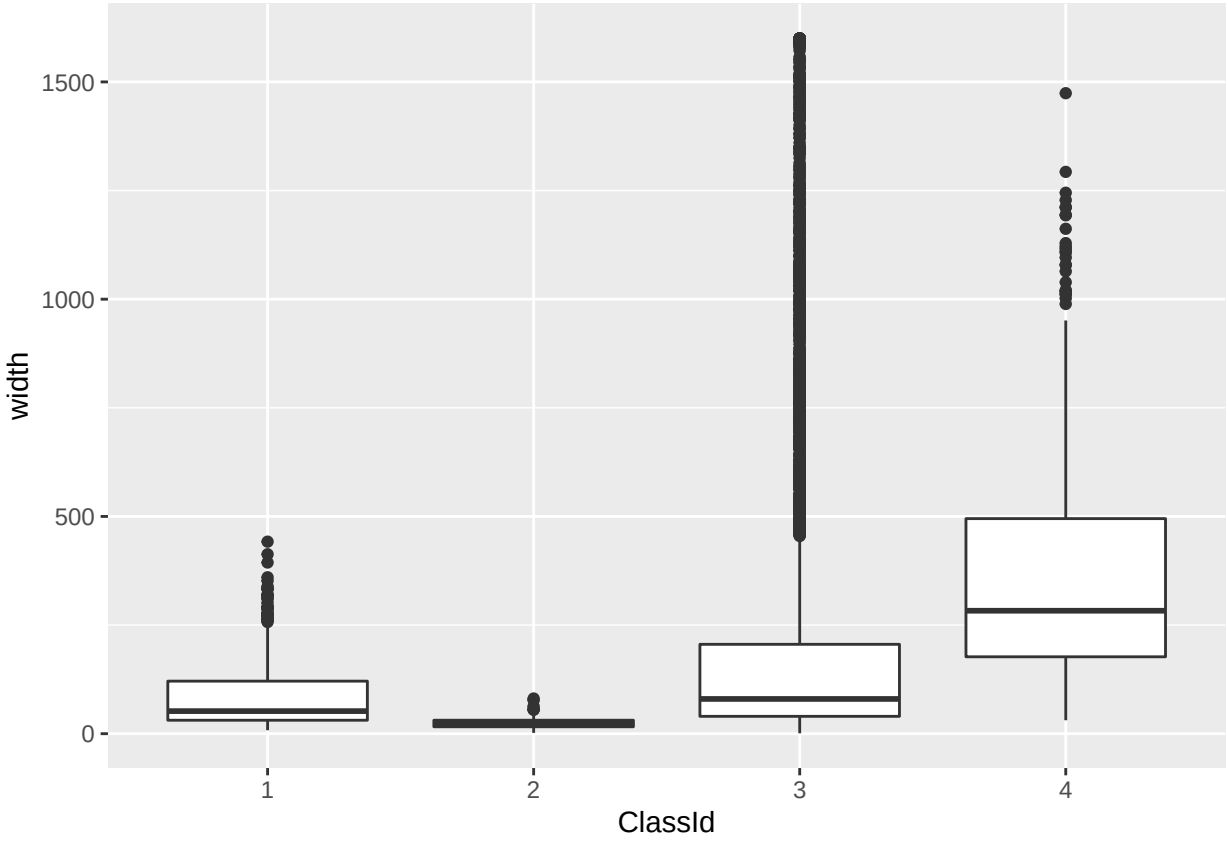

0
150
300
500
1000
1500


0
150
300
500
1000
1500



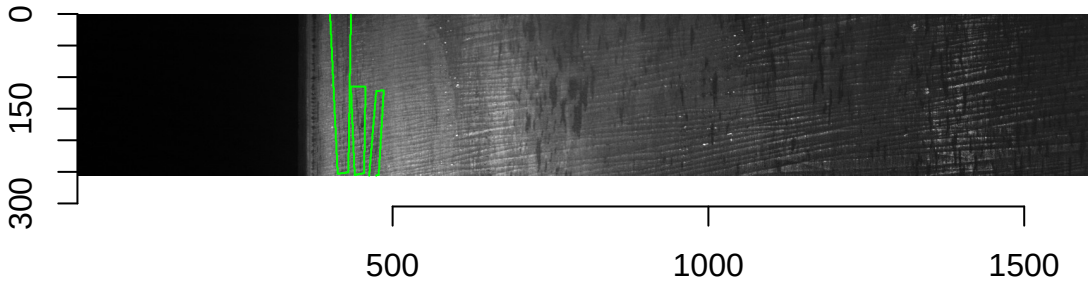
As discussed before the type 1 defects look like point defects and the outliers appear to represent a series of point defects, as there is a strong variability in greytone between small clusters of pixels.

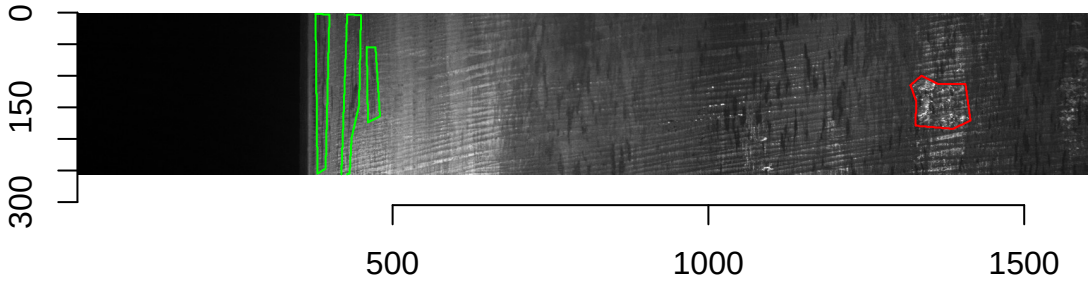
WIDTH



The boxplot for type 2 is quite remarkable, showing a very low average and variability, making it an ideal parameter to discriminate type two defects from the others. The average and the variability for defect 4 is the highest with defect three having many outliers.

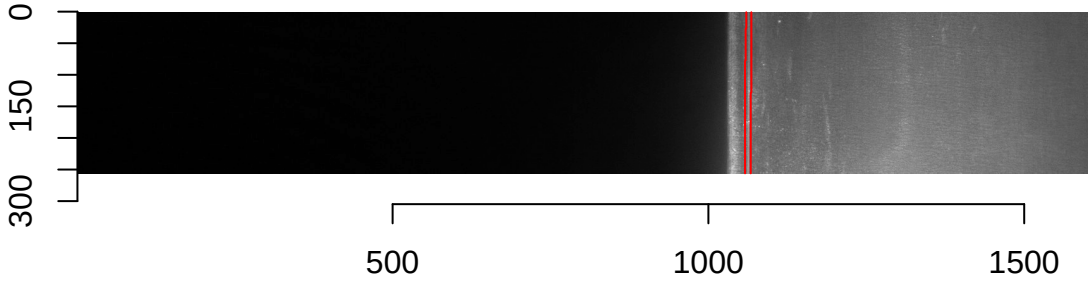
The two outliers with the highest width for defect type two are visualized underneath.

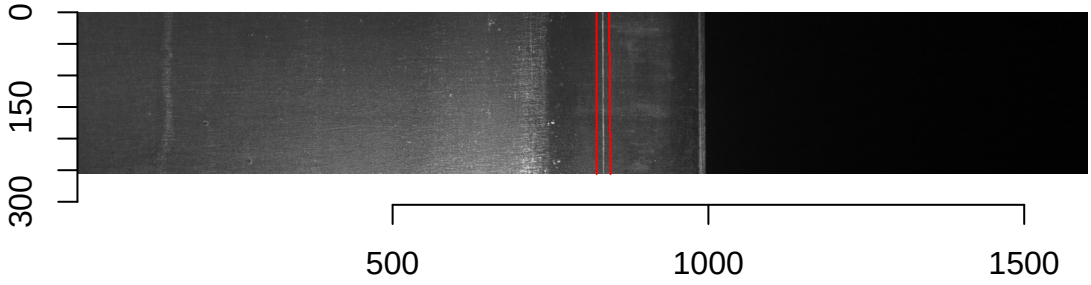


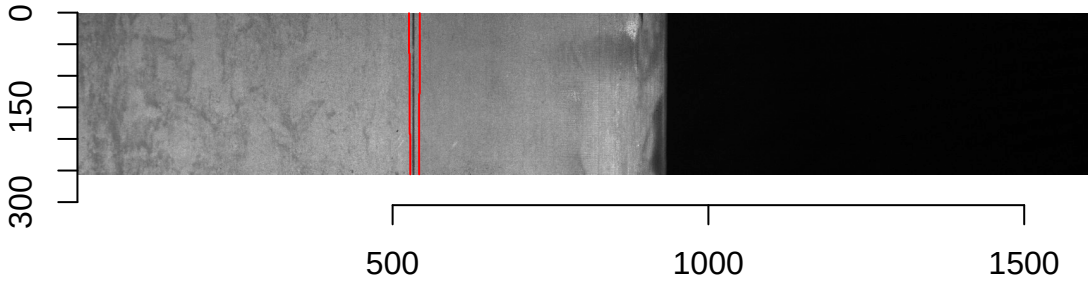


The outliers don't look any different than the sample images of type 2 defects. The reason why the width appears larger is because multiple defects of that type lie next to each other.

Also defects with a pixel width under 4 can be plotted:

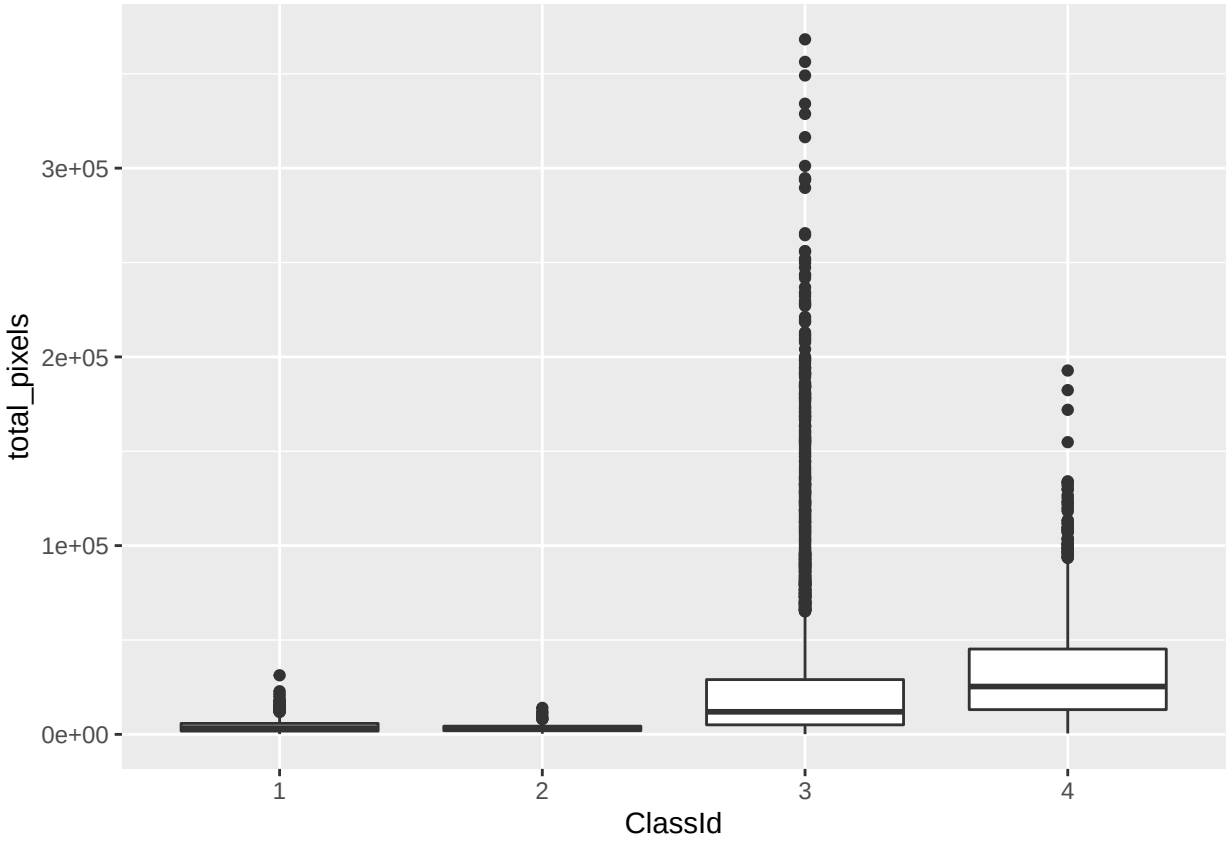

0
150
300
500
1000
1500


0
150
300
500
1000
1500



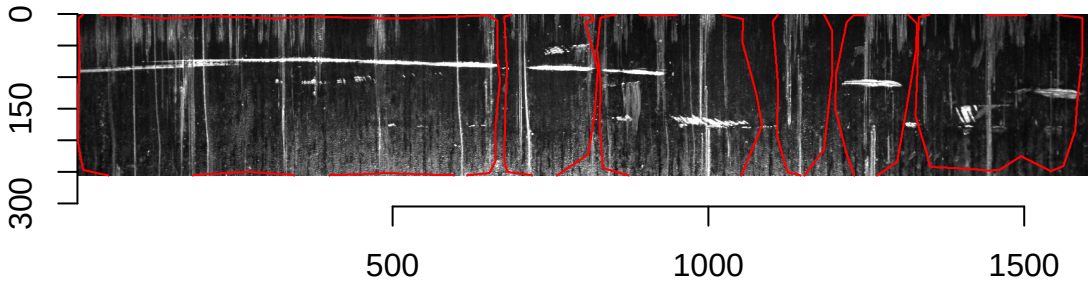
Remarkably, some defects still appear quite wide. The reason is that in the calculation of the width very long run lengths, due to defects running over the entire height of the image for mutiple pixel columns, have not been taken into account. Thus, the calculated width can be much smaller than it actually is.

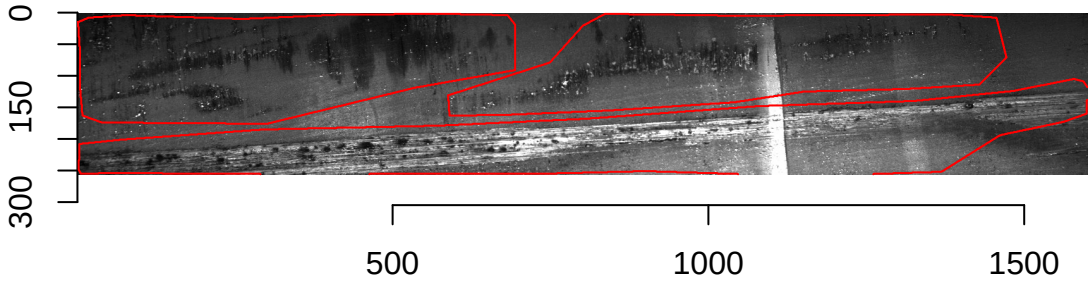
NUMBER OF PIXELS

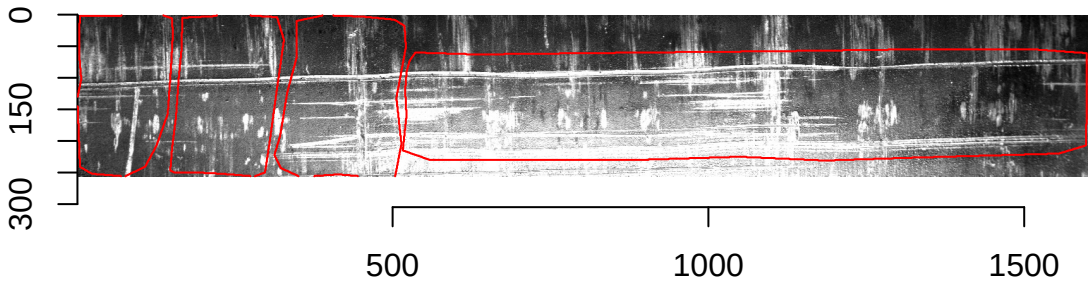


Defect 1 and defect 2 seem to be in average quite small, having a relatively small distribution. Meanwhile, defect 3 and 4 have a high average size and variability.

Some of the outliers of defect type 3 are enormous. Let's look at three of them with sizes above 300000 pixels.







These images clearly correspond to extremely damaged steel samples, as numerous scratches can be seen on the steel surface.

Conclusion

The data visualization showed that the various steel defect types have clear differences between them. From the EncodedPixels variable three variables have been extracted: width, height and size of the defects. These three have different distributions depending on the defect type. In the next chapter these variables will be used to predict the defect type present in the image. It's important to note that those variables are not completely independent from each other.

Further data visualization and wrangling efforts should focus on the number of "black", "grey" and "white" pixels within a defect since those will have even more predictive power. When possible, also the change in color within the defect should be extracted and studied.

Machine learning

In this chapter the behaviour of several popular machine learning algorithms will be tested. The confusion matrix will be shown for every algorithm to see which defect types the algorithm predict well and which not. In addition, the accuracy will be computed as a ways of benchmarking the algorithm.

First train2 has to be divided into a train set for machine learning and a test set to test the algorithm:

```
set.seed(1, sample.kind = "Rounding")
test_index <- createDataPartition(1:nrow(train2), times = 1, p = 0.1, list = FALSE)
train3 <- train2[-test_index,]
test <- train2[test_index,]

# define x (Variables) and y(ClassId)
y_train <- as.factor(train3$ClassId)
y_test <- as.factor(test$ClassId)
x_train <- as.matrix(train3[,4:6])
x_test <- as.matrix(test[,4:6])
```

The first algorithm is the naive Bayes algorithm. Abstractly, it is a form of conditional probability classifier in which the features are considered independent:

Naive Bayes

```
#naive bayes
train_naive_bayes <- train(x_train, y_train, method = "naive_bayes")
confusionMatrix(predict(train_naive_bayes, x_test), y_test)$table
```

```
##            Reference
## Prediction   1   2   3   4
##          1  49   3  84  11
##          2   9  11  15   0
##          3  25   9 375  33
##          4   1   0  35  52
```

```
confusionMatrix(predict(train_naive_bayes, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.6839888
```

It can be seen that almost every ClassId is predicted more correct than incorrect. However, there are still many incorrect predictions. The fact that the variables used in this naive bayes approach are not independent may affect the results.

QDA

QDA is a form of naive Bayes in which the features are assumed to be bivariate normal:

```
#qda
train_qda <- train(x_train, y_train, method = "qda")
confusionMatrix(predict(train_qda, x_test), y_test)$table
```

```
##            Reference
## Prediction   1   2   3   4
##          1  61   1 132  27
##          2  19  22  78   0
##          3   4   0 299  69
##          4   0   0   0   0
```

```
confusionMatrix(predict(train_qda, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.5365169
```

The qda algorithm seems to function better for ClassIds 1 and 2 with respect to naive Bayes. None of defect type 4 in the test set have been correctly predicted. Remarkably, all but one have been correctly predicted for type 2. The assumption of bivariate normality is also quite off since most boxplots shown earlier do not represent a normal distribution.

LDA

A more special case of qda is lda in which all features are assumed to yield the same correlation and standard deviations :

```
#lda
train_lda <- train(x_train, y_train, method = "lda")
confusionMatrix(predict(train_lda, x_test), y_test)$table
```

```
##            Reference
## Prediction   1   2   3   4
##          1  18   0  35   2
```

```
##          2   0   0   0   0
##          3  66  23 469  94
##          4   0   0   5   0
```

```
confusionMatrix(predict(train_lda, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.6839888
```

As can be seen, this algorithm does not appropriately predict classes 1, 2 and 4 which are almost all assigned to type 3. Most remarkably none of type 2 and type 4 have been correctly predicted. The higher accuracy is thus misleading, and originates largely from the high amount of type 3 defects in the test set and the very good prediction of type 3 defects.

knn

A completely different machine learning algorithm is the k-nearest neighbors (knn) algorithm, which classifies a particular observation by looking at the k nearest neighbours in the feature space:

```
#knn
train_knn <- train(x_train, y_train, method = "knn",
                   tuneGrid = data.frame(k = seq(3, 100, 2)))
confusionMatrix(predict(train_knn, x_test), y_test)$table
```

```
##           Reference
## Prediction   1   2   3   4
##          1  16   5  12   0
##          2   0   0   0   0
##          3  68  18 497  96
##          4   0   0   0   0
```

```
confusionMatrix(predict(train_knn, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.7247191
```

The number of neighbours k was optimized with regards to accuracy and this algorithm yields so far the best accuracy. Although, the algorithm performed very well for type 3 defects, it predicts ClassIds 1,2 and 4 rather poorly.

rpart

The rpart algorithm is a decision trees machine learning algorithm in which an observation is assigned a class by following a certain order of decisions based on the values of the features.

```
#rpart
train_rpart <- train(x_train, y_train, method = "rpart",
                     tuneGrid = data.frame(cp = seq(0.0, 0.1, len = 25)))
confusionMatrix(predict(train_rpart, x_test), y_test)$table
```

```
##           Reference
## Prediction   1   2   3   4
##          1  15   2  11   0
##          2   0   0   0   0
##          3  68  21 468  50
##          4   1   0  30  46
```

```
confusionMatrix(predict(train_rpart, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.7429775
```

This algorithm behaved even better than the knn algorithm. In addition it was also better at predicting ClassId 4. It still behaved poorly for Classid 2 and 1 though.

rf

The random forest (rf) algorithm is an extension of the decision tree algorithm, where many random decision trees are generated based on a fixed number of randomly selected features. To predict the class, an average of the decision trees is taken.

```
#rf
train_rf2 <- train(x_train, y_train, method = "rf")
```

```
## note: only 2 unique complexity parameters in default grid. Truncating the grid to 2 .
```

```
confusionMatrix(predict(train_rf2, x_test), y_test)$table
```

```
##           Reference
## Prediction   1   2   3   4
##          1  34   3  46   7
##          2   4   7   6   0
##          3  45  13 438  52
##          4   1   0  19  37
```

```
confusionMatrix(predict(train_rf2, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.7247191
```

The random forest loses a bit of its accuracy with respect to rpart, but also behaves somewhat well for other ClassIds besides type 3.

multinom

Multinomial (multinom) logistic regression is a machine learning method that generalizes logistic regression to multiple classes:

```
#multinom
train_multinom <- train(x_train, y_train, method = "multinom")
```

```
confusionMatrix(predict(train_multinom, x_test), y_test)$table
```

```
##           Reference
## Prediction   1   2   3   4
##          1  20   0  31   3
##          2   0   0   1   0
##          3  64  23 477  93
##          4   0   0   0   0
```

```
confusionMatrix(predict(train_multinom, x_test), y_test)$overall["Accuracy"]
```

```
##  Accuracy
## 0.6980337
```

It's accuracy is not as good as other methods and it was not able to predict any type 4 or type 2 defects correctly.

All models

To compare all the accuracies with one another and to make up an ensemble, all the models are also run again after each other.

```
#all models at once
models <- c("qda", "lda", "knn", "rpart", "rf",  "naive_bayes", "multinom")

set.seed(1, sample.kind = "Rounding")
```

```
fits <- lapply(models, function(model){
  print(model)
  train(x_train, y_train, method = model)
})
```

```
names(fits) <- models

predmatrix <- sapply(fits, function(x){
  predict(x,x_test)
})

pred <- map(fits, function(object) # makes the predictions for all models
  predict(object, newdata = x_test))


acc <- sapply(pred, function(object){ #accuracy of all the models
  confusionMatrix(data = object, reference = y_test)$overall[["Accuracy"]]
})

acc
```

```
##         qda         lda         knn       rpart          rf naive_bayes
##   0.5365169   0.6839888   0.6980337   0.7429775   0.7289326   0.6839888
##    multinom
##   0.6980337
```

```
#ensemble of all models

i <- c(1,2,3,4)
times_classID_predicted <- sapply(i, function(x){
  # calculates how many times an algorithm predicted a particular classId per observation
  rowSums(predmatrix == x)
})
names(times_classID_predicted) <- i
j <- seq(1,length = nrow(times_classID_predicted), by =1)
ensemble <- sapply(j, function(y){ # takes the ClassId which has been predicted the most
  as.factor(which.max(times_classID_predicted[y,]))
})
confusionMatrix(ensemble, y_test)$table
```

```
##            Reference
## Prediction   1   2   3   4
##          1  31   2  43   3
##          2   4   5   0   0
##          3  49  16 461  89
##          4   0   0   5   4
```

```r
confusionMatrix(ensemble, y_test)$overall[["Accuracy"]]
```

```
## [1] 0.7036517
```

Overall, the ensemble has a lower accuracy than the rpart or rf machine learning algorithms. There is also still a too large fraction of defect types which is incorrectly assigned to type three defects. This might happen because the data set contains many type three defects and because there is a high variability in the width, height and size of type three defects. Improvements can be made by identifying more features related to the defects and by performing cross validation.

Conclusion

In this data science project popular machine learning algorithms were applied for the classification of defects in steel samples. This project was based on a dataset containing a series of images of steel samples and a table summarizing the position and the type of defects in the steel sample images. The dataset was published on kaggle and was subject of a machine learning competition in which defects had to be determined and classified in unanalyzed steel sample images. For this capstone project the task was simplified to just classifying the type of defect based on the width, height and size of analyzed steel samples.

First a series of wrangling steps were performed to visualize the defects in the steel sample images as well as to identify new features from the information in the table containing the defect information. In particular the height, width and size were calculated. Data visualization efforts demonstrated that there were clear distinctions between the different defects regarding width, height and size. As such, machine learning algorithms to detect the class type from these features should yield better accuracies than just guessing the defect type. Indeed all tested algorithms yielded accuracies above 50 %. Some algorithms only yielded good accuracies for particular defect types, while others optimized the accuracy for all defect types, which is preferable. The rpart decision tree algorithm resulted in the highest accuracy, 74 %, but was only able to detect defect type 3 and 4 somewhat good. Meanwhile the random forest and the naive bayes algorithm were able to detect all ClassIds somehwat.

Further data science efforts for defect type classification could focus on identifying more features with regards to the pixel grayscale within the defects. In addition cross validation could be performed on the train set to have better trained algorithms. In a more advanced project algorithms could be developed to detect the position of defects in previously unanalyzed steel sample images. Since then lists of matrices with pixel grayscales have to be scanned, much higher computation power would be needed.