PH125.9x Data Science: Capstone - Casting Defect Detection

Nathan Fischer

11/14/2020

Introduction

Metal casting is a manufacturing process for producing near-net shaped metallic products by pouring molten metal into a shaped, evacuated mold. The molten metal is allowed to solidfy inside the mold, taking the shape produced by the mold. It has a wide range of applications, including jewelry, consumer products, electronic components, and aerospace. Castings are chosen over machined wrought products, such as extrusion or plate, in instances where a complex geometry makes subtractive machining operations prohibitively expensive, or impossible. One of the stigmas associated with castings is that they are perceived to be of lower quality than wrought products, due to the inherent variability associated with the process. An example of this in aviation is 14 CFR 25.621, which requires the use of casting factors for the analysis and certification of castings on transport category aircraft.

Due to this inherent variability, castings are evaluated using destrutive and non-destructive methods during the manufacturing process. This can include visual, penetrant, magnetic particle, or x-ray inspection techniques. The most common non-destructive inspection technique across all industries using castings is a visual inspection. This requires a human operator to purposefully inspect the castings for defects such as cracks, seams, porosity, inclusions, or areas where the mold did not completely fill. This can generally be done successfully when only a few castings require inspection, or the defects are sufficiently large. As the number of castings being inspected increases, or the size of the defect decreases, the probability of detecting these defects using visual inspection decreases. This study proposes the use of a machine learning algorithm to classify images of castings as acceptable or unacceptable - pass or fail - in lieu of human visual inspection.

The images used in this study are taken from the Kaggle dataset "casting product image data for quality inspection". It contains 7,348 images of submersible pump impellers, an industrial version of the "sump pump" which keeps basements from flooding during heavy rain. The images have already been classified by a human operator as either "defective" or "ok". In this study, the images will be imported into R and convereted into a format suitable for classification algorithms. The data will then receive some pre-processing before being evaluated using several algorithms. The most promising algorithms will then be selected for parameter optimization and final comparison.

Analysis

Importing the Data

The initial step is to download the Kaggle dataset and save it in a folder called *data*. For convenience, a copy of the dataset is saved in the associated "GitHub repository". The dataset has already been split into *test* and *train* folders by the originator, with images placed in either the *def_front* or *ok_front* sub-folders. To obtain an understanding of a typical image in the dataset, the following code was used:

```
image <- load.image("data/test/ok_front/cast_ok_0_10.jpeg")
dim(image)
```

```
## [1] 300 300   1   3
```

```
image
```

```
## Image. Width: 300 pix Height: 300 pix Depth: 1 Colour channels: 3
```

```
plot(image, axes = FALSE)
```

From this, it is discovered that the images are 300 x 300 pixel images in color of the impeller casting. It is suprising that the image has three color channels (RGB), as the image appears to be greyscale.

To import these images, a function was developed to create a list of all images in a particular folder, convert each image into greyscale, and then scale down each image to 10% of the original resolution, that is, 30 x 30 pixels instead of the original 300 x 300 pixels, for ease of modeling. Each image is converted into a row of numeric values between 0 and 1 for each pixel in the image. Each row is then added to a matrix, where each row represents a separate image in the dataset.

```
scale=0.10
get_image_matrix <- function(path,scale){
    files <- list.files(path=path, pattern=".jpeg",all.files=T, full.names=T, no.. = T)
    list_of_images <- map(files, function(.x){
      image <- load.image(.x)
      grey_image <- grayscale(image)
      grey_image_small <- imresize(grey_image,scale=scale)
    })
    image_matrix = do.call('rbind', map(list_of_images, as.numeric))
    return(image_matrix)
}
```

To import the test dataset, the **get_image_matrix** function is applied to both the *ok_front* and *def_front* sub-folders within *test*. In order to create a corresponding dependent variable, an array of the text "pass" with the same number of entries as the number of rows in the matrix created by the **get_image_matrix** function applied to the *ok_front* folder was created. A similar array of the text "fail" was created for the matrix corresponding to the *def_front* folder. The pass and fail matrices are then joined together to create a single test input matrix, while the arrays of "pass" and "fail" are also joined and turned into a factor.

```
x_test_pass <- get_image_matrix(path="data/test/ok_front",scale=scale)
y_test_pass <- array(data=rep.int("pass",nrow(x_test_pass)))

x_test_fail <- get_image_matrix(path="data/test/def_front",scale=scale)
y_test_fail <- array(data=rep.int("fail",nrow(x_test_fail)))

x_test <- rbind(x_test_pass,x_test_fail)
y_test <- factor(array(c(y_test_pass,y_test_fail)))
```

To import the train dataset, a similar process is used.

```
x_train_pass <- get_image_matrix(path="data/train/ok_front",scale=scale)
y_train_pass <- array(data=rep.int("pass",nrow(x_train_pass)))

x_train_fail <- get_image_matrix(path="data/train/def_front",scale=scale)
y_train_fail <- array(data=rep.int("fail",nrow(x_train_fail)))

x_train <- rbind(x_train_pass,x_train_fail)
y_train <- factor(array(c(y_train_pass,y_train_fail)))
```

After importing the data is complete, it is important to verify that it was imported properly.

```
dim(x_train_pass)
```

```
## [1] 2875 1369
```

```
dim(x_train_fail)
```

```
## [1] 3758 1369
```

```
dim(x_train)
```

```
## [1] 6633 1369
```

```
str(x_train)
```

```
##  num [1:6633, 1:1369] 0.601 0.649 0.521 0.611 0.886 ...
```

```
length(y_train)
```

```
## [1] 6633
```

```
str(y_train)
```

```
##  Factor w/ 2 levels "fail","pass": 2 2 2 2 2 2 2 2 2 2 ...
```

```
table(y_train)
```

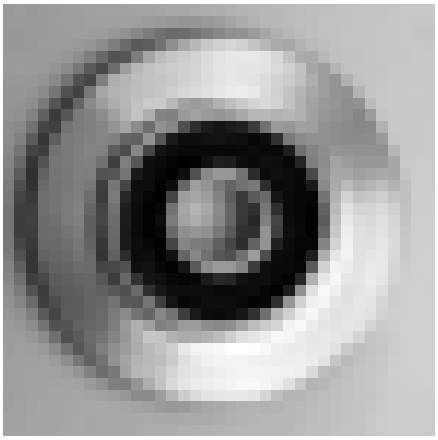
```
## y_train
## fail pass
## 3758 2875
```

Importing the data appears to be successful. The number of columns in x_train_pass, x_train_fail, and x_train all match and the number of rows in x_train is the sum of the number of rows in x_train_pass

and x_train_fail. The number of rows in x_train also matches the number of rows in y_train, which was correctly imported as a factor with two levels, with the first being "fail" and the second being "pass". The number of "fail" entries in y_train matches the number of rows in x_train_fail and the number of "pass" entries matches the number of rows in x_train_pass.

It is suprising how balanced the dataset is between the "pass" and "fail" conditions. One would expect a profitable foundry to have notably lower instances of unacceptable castings than acceptable castings. Due to this lack of bias in the dataset, overall accuracy will be used for the initial model selection.

The final check before progressing onto pre-processing is to view an original image and visually compare it to the same image after being imported and resized.



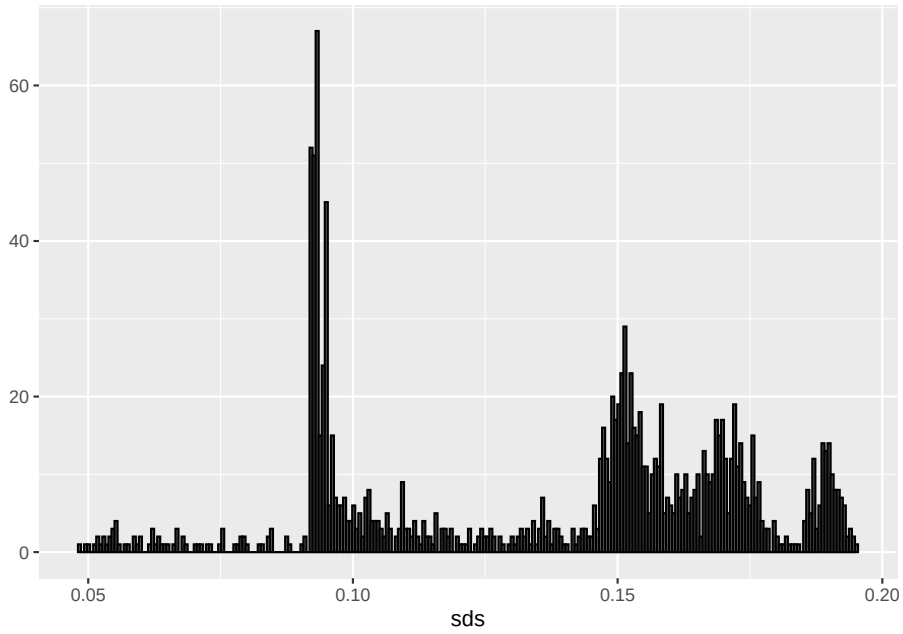
It can be seen that the imported version of this image can be identifed as the same impeller, with matching locations of shadowing, except the imported version is a lower resolution image. All of these results provide confidence that the image importing technique used in this study was successful.

Pre-Processing

Pre-processing is the term used for data transformations performed on the dataset prior to using classification or regression algorithms. Common techniques include centering and scaling the data, entering missing data points, transforming predictors, or creating dummy variables for categorical data. These steps are performed for various reasons, including to improve accuracy, reduce runtime, or even to allow the algorithm to function at all.

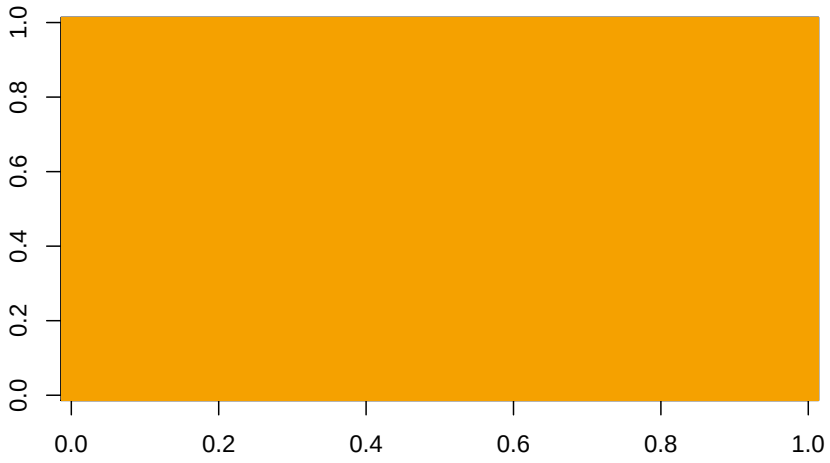
For this image-based dataset, a useful pre-processing step is to remove any predictors - columns - in the dataset that have zero or near-zero variance. This correlates to locations in the images which are nearly identical and therefore do not provide any useful data regarding the acceptability of a casting. A plot of the standard deviations of the columns of the training dataset is created using the following code.

```
sds <- colSds(x_train)
qplot(sds, bins = 256, color = I("black"))
```



Visually, there does not appear to be any predictors with near zero variance. To verify this, the **nearZeroVar** function is used, with an image created to show the locations in the image dataset identified as having near zero variance.

```
nzv <- nearZeroVar(x_train)
image(matrix(1:ncol(x_train) %in% nzv, image_dim, image_dim))
```




```
col_index <- setdiff(1:ncol(x_train), nzv)
length(col_index)
```

```
## [1] 1369
```

The image produced is blank and the number of items in col_index (1369) matches the number of columns

in the original dataset. The results of using the **nearZeroVar** function matches the visual observation from the plot of column standard deviations, that is, all of the predictors are providing useful information for classifying the images. Even though all of the columns were included in col_index, the index will continue to be used in the present study, as future work that increases the resolution of the imported images from the current ten percent may find removing predictors with near zero variance valuable.

The final pre-processing step is to add column names to the dataset, as required by the **caret** function.

```
colnames(x_train) <- 1:ncol(x_train)
colnames(x_test) <- colnames(x_train)
```

Model Selection

To select which models to use for parameter optimization, several common models will be fit to the dataset using default parameters. This will include Naive Bayes, Quadratic Discriminant Analysis, Linear Discriminant Analysis, Stochastic Gradient Boosting, k-Nearest Neighbors, Random Forest, and eXtreme Gradient Boosting. To minimize processing time, two-fold cross validation will be used.

```
models <- c("naive_bayes","qda","lda","gbm","knn","rf","xgbDART")
set.seed(1, sample.kind = "Rounding")
control <- trainControl(method = "cv", number = 2, p = 0.8)
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)
fits <- map_df(models, function(model){
  ptm <- proc.time()
  fit<-train(x = x_train[, col_index],y = y_train, method = model, trControl = control)
  ptm <- proc.time() - ptm
  tibble(model=model,
         time=ptm[[3]],
         min=min(fit$results$Accuracy),
         max=max(fit$results$Accuracy),
         avg=mean(fit$results$Accuracy)
         )
})
stopCluster(cl)
```

To evaluate the results from each of the models, the processing time as well as the minimum, maximum, and average accuracy is compared.

model	time	min	max	avg
rf	2100.16	0.9656263	0.9684909	0.9670837
knn	419.92	0.9503998	0.9562793	0.9531637
xgbDART	1134.75	0.7566714	0.9781394	0.9298523
gbm	152.14	0.8744167	0.9553752	0.9242679
naive_bayes	95.33	0.7979807	0.8643155	0.8311481
lda	173.45	0.8249666	0.8249666	0.8249666
qda	108.36	0.5668627	0.5668627	0.5668627

The models with the highest accuracy are Random Forest, k-Nearest Neighbor, eXtreme Gradient Boosting, and Stochastic Gradient Boosting. Based on the high accuracy and low processing time, Stochastic Gradient Boosting and k-Nearest Neighbors were selected for parameter optimization.

Stochastic Gradient Boosting Model

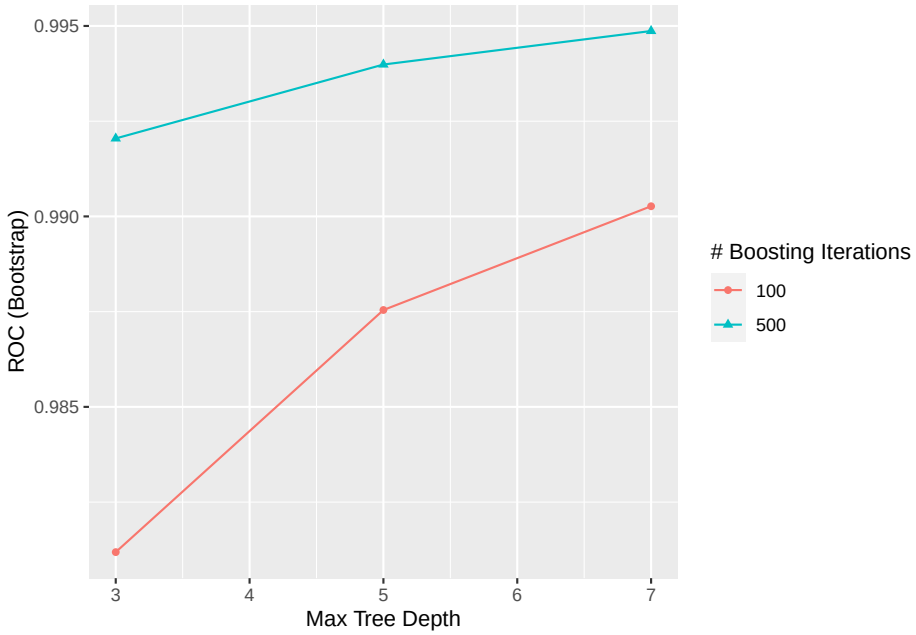
The first model to be optimized is Stochastic Gradient Boosting, which includes the tuning parameters n.trees, interaction.depth, shrinkage, and n.minobsinnode. To prevent overfitting, 10-fold cross validation using the bootstrap method is used.

Another change from the process used for model selection is that accuracy will not be used to optimize parameters or to select the optimal model. This is due to the differences in costs associated with the model correctly or incorrectly determining a "pass" or "fail" in this application. It is far more costly, in terms of revenue, reputation, etc, for a true "fail" to be predicted as "pass"and sent to the customer, than it is for a true "pass" to be predicted as "fail" and scrapped. Because of this, a method of evaluating performance other than accuracy is needed. Since "fail" comes before "pass" in the alphabet, "fail" is considered the positive outcome in **confusionMatrix**, so maximizing sensitivity will be the objective of this investigation. This can be implemented on a binary classification problem by changing summaryFunction to twoClassSummary and metric to ROC. The best model will then be selected which maximizes the area under the ROC curve, a trade-off between sensitivity and specificity.

The following code is used to fit the model to the training dataset.

```
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)
set.seed(1, sample.kind = "Rounding")
control <- trainControl(method = "boot", number = 10, classProbs=TRUE,
                        summaryFunction = twoClassSummary)
tune <- expand.grid(interaction.depth=c(3,5,7), n.trees=c(100,500),
                    shrinkage=c(0.1), n.minobsinnode=10)
ptm <- proc.time()
train_gbm <- train(x_train[, col_index], y_train,
                   method = "gbm",
                   metric = "ROC",
                   tuneGrid = tune,
                   trControl = control)
stopCluster(cl)
```

```
ggplot(train_gbm)
```




```
train_gbm$bestTune %>% kable()
```

	n.trees	interaction.depth	shrinkage	n.minobsinnode
6	500	7	0.1	10

Looking at the selected optimal parameters, the maximum number of trees (n.trees) and interaction depth (interaction.depth) were found to be optimal, with the area under the ROC curve approaching 0.995. It is likely that increaseing these values could improve prediction results at the cost of increased processing time.

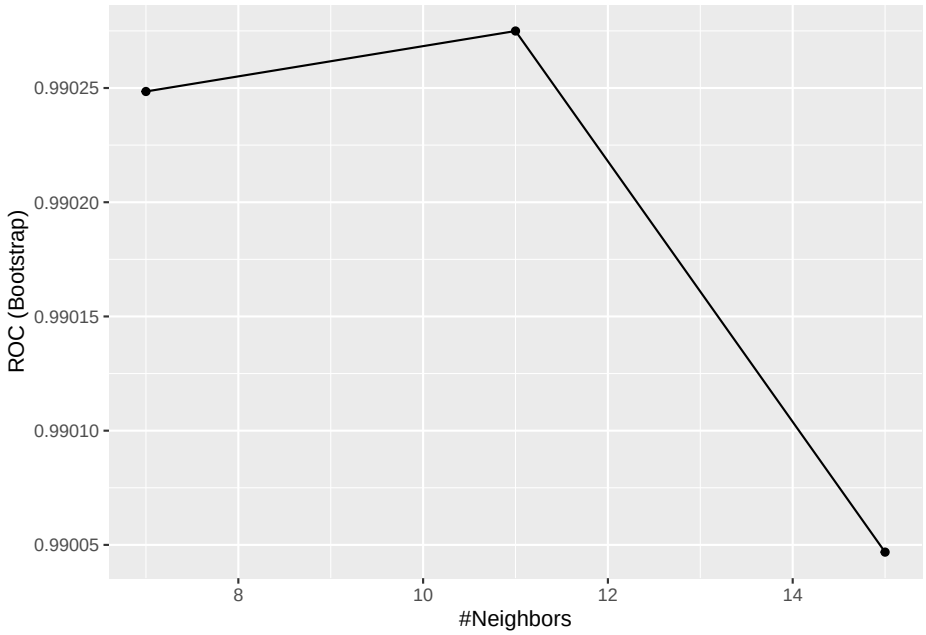
K-Nearest Neighbors Model

The next model to be optimized is k-Nearest Neighbors, which includes a single tuning parameter, k. Once again, 10-fold cross validation using the bootstrap method is used to select the optimal k value.

The following code is used to fit the model to the training dataset.

```
cl <- makePSOCKcluster(detectCores())
registerDoParallel(cl)
ptm <- proc.time()
set.seed(1, sample.kind = "Rounding")
control <- trainControl(method = "boot", number = 10, classProbs=TRUE,
                        summaryFunction = twoClassSummary)
train_knn <- train(x_train[, col_index], y_train,
                   method = "knn",
                   metric = "ROC",
                   tuneGrid = data.frame(k = c(7,11,15)),
                   trControl = control)
stopCluster(cl)
```

```
ggplot(train_knn)
```




```
train_knn$bestTune %>% kable()
```

	k
2	11

The results show that the optimal number of neighbors, k, is around 11, with an area under the ROC curve slightly over 0.99. This is lower than the results from Stochastic Gradient Boosting, but is still an excellent initial result. It is encouraging that the results did not show improved performance as the value of k decreased. If the predicted results had been maximized with k = 1, where the estimate in the training set is determined by the corresponding output for that row, it is evidence for over-training and the model would likely not have performed well on the test dataset.

Results

As a final analysis of the two models, the trained and optimized models are used to predict the outcome for the test dataset. As stated earlier, sensitivity is the primary performance metric in this situation, as allowing unacceptable product to be sent to the customer has a higher cost than having acceptable product scrapped. In addition to sensitvity, the accuracy, and F_1-score (balanced accuracy) are compared.

For the Stochastic Gradient Boosting model, the following code was used to make predictions on the test dataset.

```
y_hat_gbm <- predict(train_gbm,
                     x_test[, col_index]
                     )
cm_gbm <- confusionMatrix(data = y_hat_gbm, reference = y_test)
results <- tibble(Method="GBM",
                  Sensitivity = cm_gbm$byClass["Sensitivity"],
                  F1 = cm_gbm$byClass["F1"],
                  Accuracy = cm_gbm$overall["Accuracy"]
                  )
```

For the k-Nearest Neighbors model, the following code was used to make predictions on the test dataset.

```
y_hat_knn <- predict(train_knn,
                     x_test[, col_index]
)
cm_knn <- confusionMatrix(data = y_hat_knn, reference = y_test)
results <- bind_rows(results,
                     tibble(Method="KNN",
                            Sensitivity = cm_knn$byClass["Sensitivity"],
                            F1 = cm_knn$byClass["F1"],
                            Accuracy = cm_knn$overall["Accuracy"]
                            )
                     )
```

The results table contains the performance metrics necessary to evaluate the performance of the two models.

Method	Sensitivity	F1	Accuracy
GBM	0.9867550	0.9911308	0.9888112
KNN	0.9558499	0.9730337	0.9664336

Across all performance metrics - sensitivity, F_1-score, and accuracy - the Stochastic Gradient Boosting model provided superior performance, with sensitivy approaching 99%.

Conclusion

The goal of this investigation was to create a machine learning model that could accurately predict the acceptability or rejection of a submersible pump impeller casting based on images. After successfully importing the images and converting them into a format suitable for machine learning algorithms, multiple classification models were evaluated based on accuracy and computation time. From this, Stochastic Gradient Boosting and k-Nearest Neighbors were selected for parameter optimization. With area under the ROC curve as the primary performance metric, optimized models were created and used to make predictions on the test dataset, which was not used for model optimization. Comparing the two optimized models, the Stochastic Gradient Boosting model proved to be superior, with sensitivity approaching 99%.

While this investigation was able to achieve high performance, there are limitations that will curb its practical usefulness. This model is only applicable to a single casting design with a particular setup for taking images. Images of the same castings taken from different orientations with different lighting or working distances will likely have impaired performance compared to this test case. This model will also not likely create correct predictions on images of other castings produced at the foundry or other submersible pump impeller models, as they were not part of the training dataset.

Performance for the current Stochastic Gradient Boosting could be marginally improved by increasing the maximum number of trees and the interaction depth, with the associated trade-off of increased computational time. Other potential future work includes training optimized models using Random Forest and eXtreme Gradient Boosting methods. Both of these algorithms had high accuracy numbers during model selection, but were not used due to their long computational time. With these optimized models, an ensemble of the four models could also be evaluated for increased peformance. More advanced methodologies, such as Keras and Tensorflow could also be investigated for applicability.