Boosting for Bearing Fault Classification

Biswajit Sahoo

Boosting is slow learning procedure. In boosting, a tree is not allowed to grow fully. Rather, the depth is fixed to a small number. In some cases, a tree is build with only one branch and two nodes. But subsequent trees are built on top of it. Further, contributions from subsequent trees are also shrinked and added to previously built tree. Though an individual tree might preform badly for the classification, combination of many trees perform surprisingly well. For more details regarding the algorithm, readers can refer to this excellent book.

In this post we will apply boosting to classify multiclass bearing fault.

Description of data

Detailed discussion of how to prepare the data and its source can be found in this post. Here we will only mention about different classes of the data. There are 10 classes and data for each class are taken at a load of 1hp. The classes are:

- C1 : Ball defect (0.007 inch)
- C2 : Ball defect (0.014 inch)
- C3 : Ball defect (0.021 inch)
- C4 : Inner race fault (0.007 inch)
- C5 : Inner race fault (0.014 inch)
- C6 : Inner race fault (0.021 inch)
- C7 : Normal
- C8 : Outer race fault (0.007 inch, data collected from 6 O'clock position)
- C9 : Outer race fault (0.014 inch, 6 O'clock)
- C10 : Outer race fault (0.021 inch, 6 O'clock)

Codes

```
library(reticulate)
use_condaenv("r-reticulate")
```

First download the data from here. Save the data in a folder and read it from that folder.

```
data_wav_energy = read.csv('feature_wav_energy8_48k_2048_load_1.csv',
                           header = T)
# Change the above line to include your folder that contains data
set.seed(1)
index = c(sample(1:230,75),sample(231:460,75), sample(461:690,75),
          sample(691:920,75),sample(921:1150,75),sample(1151:1380,75),
          sample(1381:1610,75),sample(1611:1840,75),sample(1841:2070,75),
          sample(2071:2300,75))

train_data = data_wav_energy[-index,]
test_data = data_wav_energy[index,]

# Shuffle data
```

```
train_data = train_data[sample(nrow(train_data)),]
test_data = test_data[sample(nrow(test_data)),]
```

It should be noted that for some of the deterministic techniques, shuffling of data is not required. But some other techniques like deep learning require the data to be shuffled for better training. So as a recipe we always shuffle data whether the method is deterministic or not. This doesn't hurt either for a deterministic technique.

We will perform gradient boosting using 'gbm' package.

```
library(gbm)
```

```
## Loaded gbm 2.1.5
```

```
boosting_fit = gbm(fault~., train_data, distribution = "multinomial",
                   interaction.depth = 2, n.trees = 500)
pred_boosting = predict(boosting_fit, test_data,n.trees = 500,
                        type = "response")
dim(pred_boosting)
```

```
## [1] 750  10   1
```

Note that boosting prediction is in terms of a matrix. The rows correspond to data points and columns correspond to fault classes. Each row of this matrix gives the probability of the observation being in the class corresponding to the column. The data point is classified to a category for which it has highest probability of occurrence. This can be done by the following code

```
prediction = colnames(pred_boosting)[apply(pred_boosting, 1, which.max)]
# Confusion matrix
test_confu = table(test_data$fault, prediction)
```

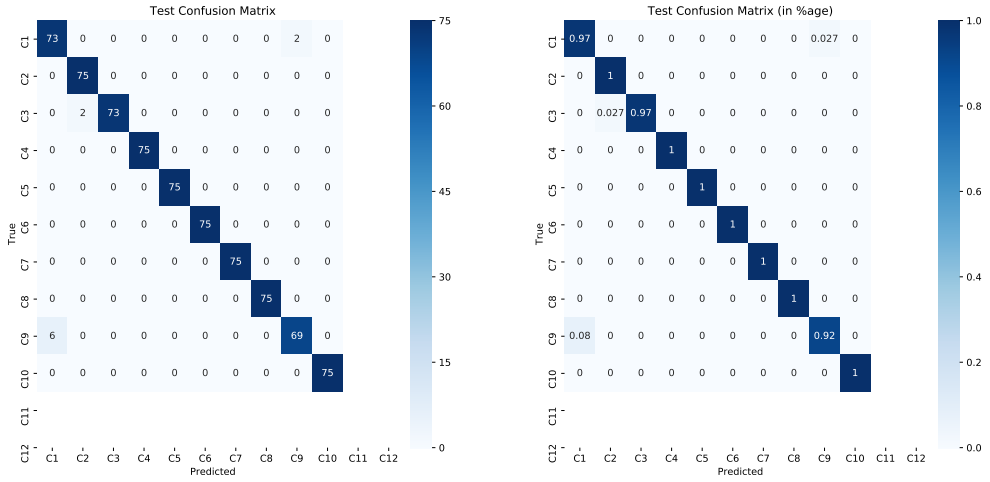
```
import seaborn as sns
import matplotlib.pyplot as plt
fault_type = ['C1','C2','C3','C4','C5','C6','C7','C8','C9','C10','C11','C12']
plt.figure(1,figsize=(18,8))
plt.subplot(121)
sns.heatmap(r.test_confu, annot = True,
xticklabels=fault_type, yticklabels=fault_type, cmap = "Blues")
```

```
## <matplotlib.axes._subplots.AxesSubplot object at 0x000000002652FDA0>
```

```
plt.title('Test Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.subplot(122)
sns.heatmap(r.test_confu/75, annot = True,
xticklabels=fault_type, yticklabels=fault_type, cmap = "Blues")
```

```
## <matplotlib.axes._subplots.AxesSubplot object at 0x0000000028C92FD0>
```

```
plt.title('Test Confusion Matrix (in %age)')
plt.xlabel('Predicted')
plt.ylabel('True')
plt.show()
```




```
overall_test_accuracy = sum(diag(test_confu))/750
sprintf("Overall Test Accuracy: %.4f", overall_test_accuracy*100)
```

```
## [1] "Overall Test Accuracy: 98.6667"
```

There are several hyper-parameters in this model, number of trees, interaction depth, and shrinkage. Optimal values for each of these parameters can be obtained by cross validation. In this post, we have chosen some of the commonly used values. Other values can also be tried.

To see results of other techniques applied to public condition monitoring datasets, visit this page.

Last updated: 8^{th} July, 2019