Decision Tree for Bearing Fault Classification

Biswajit Sahoo

Most classification algorithms form a model (some sort of functional mapping) that maps input data to output. But decision trees adopt a different strategy. Decision trees divide the input space into regions and do classification based on majority vote in each region. Let's consider an example that has a single predictor and the target values are either 0, 1, or 2. Most of the times there is no clear separation of labels in terms of predictor values. In that case, a predictor value is chosen that maximizes the information gain. If the predictor varies between (0,100), first separation can be made anywhere between 1 to 99. So information gain is computed for all possible separation points starting form 1 to 99. The first cut is made at a value for which information gain is maximum.

Calculation of information gain is not direct. Rather, information gain depends on impurity function of each group that appear after separation. There are several choices for impurity function. We will not go into the details of it here. We will save that for a later time. For the time being interested readers can refer to this excellent short paper or this excellent book.

We will show results of applying decision tree to condition monitoring data using R codes.

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.

```
library(tree)
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.

```
fit_tree = tree(fault~., train_data)
pred_tree =predict(fit_tree, test_data, type = "class")
# Confusion matrix
test_confu = table(test_data$fault, pred_tree)
```

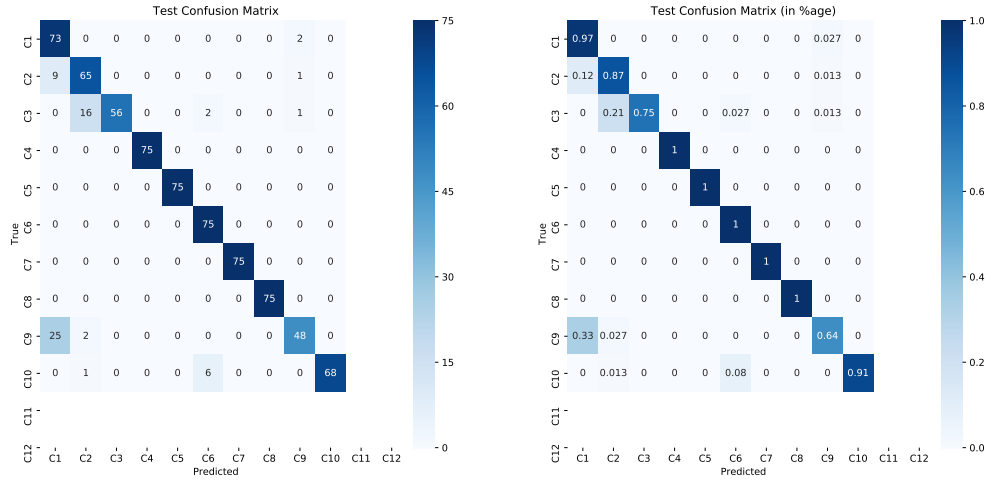
```
import seaborn as sns
import matplotlib.pyplot as plt
fault_type = ['C1','C2','C3','C4','C5','C6','C7','C8','C9','C10','C11','C12']
plt.figure(1,figsize=(18,8))
plt.subplot(121)
sns.heatmap(r.test_confu, annot = True,
xticklabels=fault_type, yticklabels=fault_type, cmap = "Blues")
```

```
## <matplotlib.axes._subplots.AxesSubplot object at 0x0000000022C6E438>
```

```
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 0x0000000024C70128>
```

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

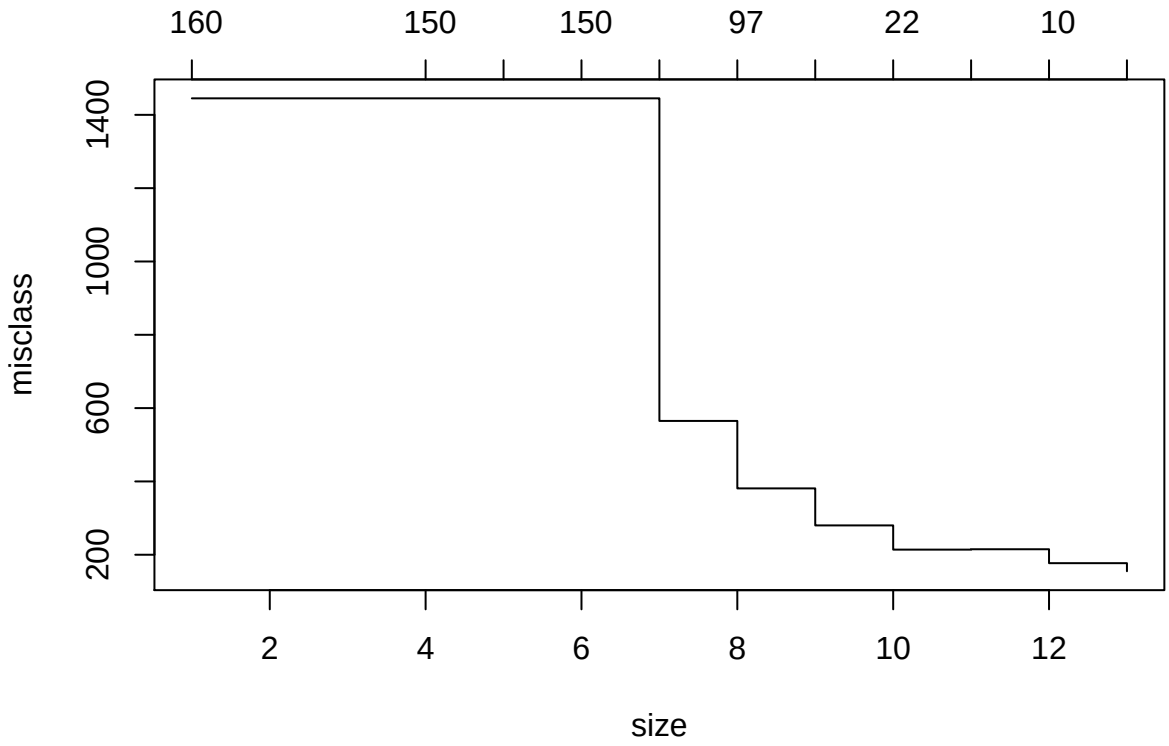



```
overall_test_accuracy = sum(diag(test_confu))/750
sprintf("Overall Test Accuracy: %.4f", overall_test_accuracy*100)
```

```
## [1] "Overall Test Accuracy: 91.3333"
```

Usually, the default tree grown by the algorithm contains many nodes and generalizes poorly. One strategy to overcome this is to prune the tree at certain number of nodes. The optimal cut size can be obtained from cross validation.

```
set.seed(101)
tree_fit_cv = cv.tree(fit_tree, FUN = prune.misclass)
plot(tree_fit_cv)
```



As the lowest misclassification accuracy is obtained when the tree is completely grown, we don't prune the tree.

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

Last updated: 8^{th} July, 2019