Data Science Development Coursework (CMM535) Comparitive experiments using PCA as a noise reduction technique

Owen Jenkins, *o.Jenkins@rgu.ac.uk*

April 29, 2019

Contents

1 Introduction

The accuracy of ultrasonic flow meters enables many industrial processes to be operated in a safe manner using automatic control systems, requiring minimal human intervention. They rely on the well understood physics of sound wave propogation in fluids, first published by John William Rayleigh in 1877 [1]. The measurement relies on the principle that sound waves will travel faster going with the direction of flow and slower going against the flow as shown in figure 1. At T1 an ultrasonic pulse is transmitted which bounces off the pipe wall and is received at T2 at a known distance from the transmitter, the time difference for the transmission and reception of these pulses can then be used to calculate the speed of the fluid in the pipe.

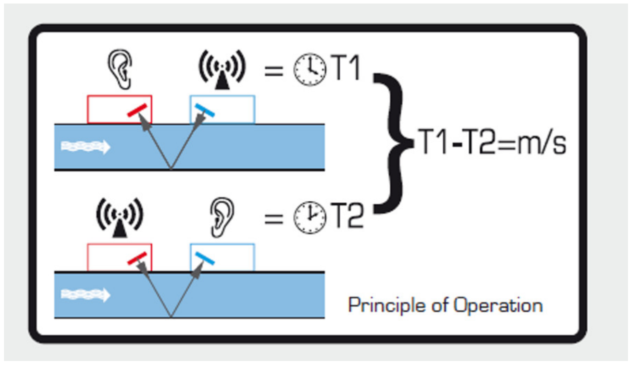


Figure 1: Ultrasonic flowmeter principle of operation [2]

In practice there is always a noise component to these measurements in real systems, due to vibrations from nearby equipment or echo effects in the pipe, causing overfit in classification algorithms. These experiments use Principal Component Analysis as an exploratory and preprocessing technique for the items listed below:

1. Reduce noise in the system to avoid overfitting
2. Mitigate multicollinearities between variables
3. Reduce processing time

The dataset used in this experiment was obtained from the UCI machine learning repository http://archive.ics.uci.edu/ml/datasets/Ultrasonic+flowmeter+diagnostics, where the authors propose a similar study using a different dimensionality reduction technique [3].

2 Exploratory analysis

Investigation of the dataset using the code below identified 43 numerical predictive attributes relating to the ultrasonic and physical properties of the fluid (see section 6.1 for full attribute descriptions and layout diagram) with no missing attributes.

```
#check dimensions, attribute names, missing values and head sample of dataframe
names(df)
colSums(sapply(df, is.na))
```

The class attribute consisted of a vector with integer numbers of 1 to 4 representing the operating state of the flowmeter for each instance.

1. Healthy
2. Waxing
3. Gas injection
4. Installation effects

2.1 Class attribute distribution

The distribution of the class variable "Meter_State" shows that there are roughly 50% fewer instances showing wax deposition on the meter, likely due to waxing only being observed at start up from a clean state. Owing to this imbalance, a bootstrap is suggested over a stratified random sampling technique, to ensure that the underlying class attribute distribution is preserved and there are sufficient instances of the wax deposition class in the training dataset.

```
#Plot class attribute distribution
g <- ggplot(data = df, aes(x = Meter_State)) +
  geom_bar(stat = "count", fill = "green") +
  theme_light() +
  labs(title = "Meter state distribution") +
  theme(plot.title = element_text(hjust = 0.5))
g
```

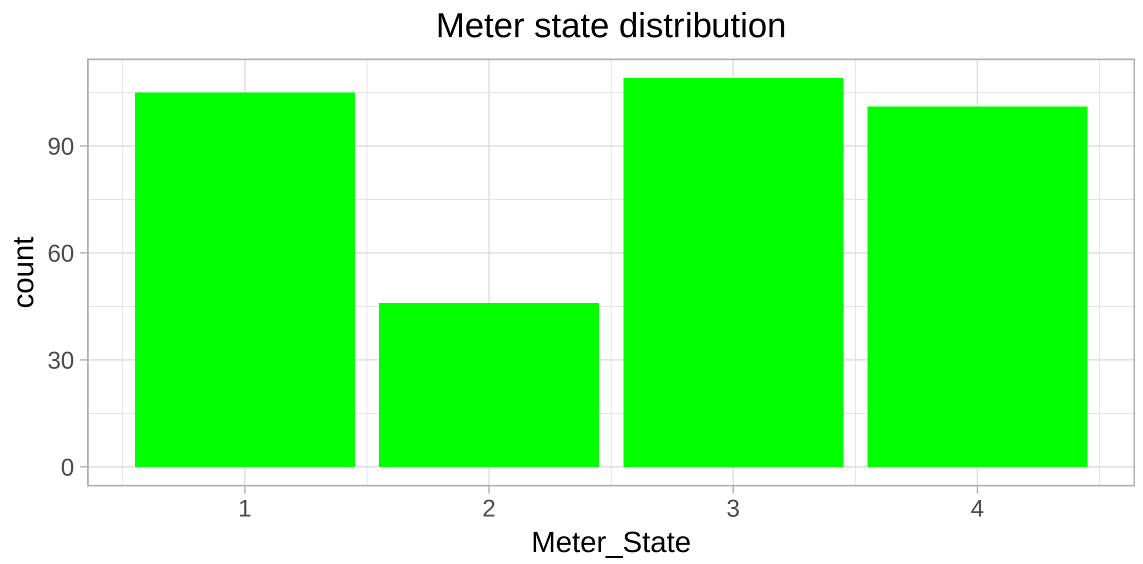


Figure 2: Meter State distribution

2.2 Correlation Matrix

A correlation matrix of the predictive attributes indicated that there was strong collinearity between attributes belonging to the same subgroup. This is unsurprising as each of the attributes within groups will depend on the same physical variables of the fluid, such as the density. The strong multicollinearity supports the use of PCA to preprocess the data to guarantee independence between the attributes.

```r
#calculate Pearsons correlation coefficients for data frame,
#marking uncorrelated variables with an X at 95% confidence
res1 <- cor.mtest(df, conf.level = 0.95)

#Generate correlation matrix for data frame
correlation_matrix <- corrplot(corr = cor(df),
                               p.mat = res1$p,
                               method = "color",
                               tl.col = "black",
                               tl.srt = 45,
                               type = "lower",
                               tl.cex = 0.5)
```

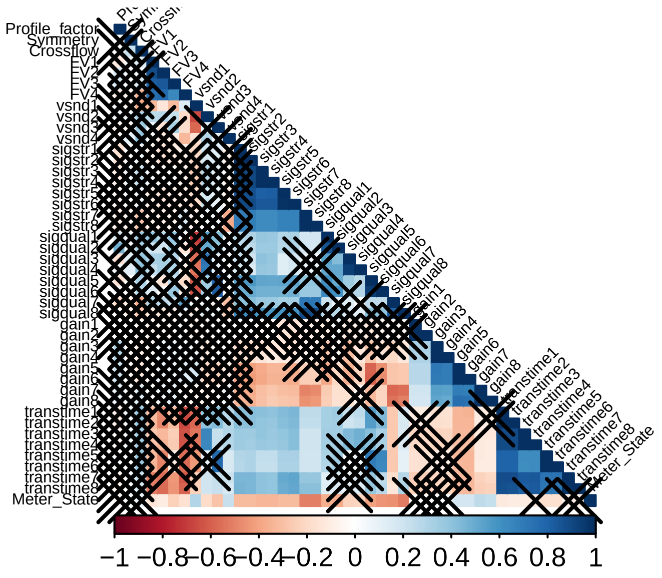


Figure 3: Pearson correlation matrix

2.3 Boxplot of variables

To inspect the distribution of variables across the class attribute, the following functions were used:

```r
plotbxp <- function(data_in, i, j = ncol(data_in)) {
  p <- ggplot(data_in, aes(x=data_in[[j]], y=data_in[[i]], fill = data_in[[j]])) +
    stat_boxplot(geom = 'errorbar') + geom_boxplot() +
    xlab(colnames(data_in)[j]) + ylab(colnames(data_in)[i]) +
    stat_summary(fun.y=mean, colour="darkred", geom="point", hape=18,
                 size=3,show_guide = FALSE) +
    stat_summary(fun.y=mean, colour="red", geom="text", show_guide = FALSE,
                 vjust=-0.7, aes( label=round(..y.., digits=1))) +
    theme(legend.position = "none")


  return (p)
}

doPlots <- function(data_in, fun, ii, ncol=3) {
  pp <- list()
  for (i in ii) {
    p <- fun(data_in=data_in, i=i)
    pp <- c(pp, list(p))
  }
  do.call("grid.arrange", c(pp, ncol=ncol))
}

doPlots(df, fun = plotbxp, ii = 1:3, ncol = 2)
doPlots(df, fun = plotbxp, ii = 4:7, ncol = 2)
doPlots(df, fun = plotbxp, ii = 8:11, ncol = 2)
doPlots(df, fun = plotbxp, ii = 12:19, ncol = 4)
doPlots(df, fun = plotbxp, ii = 20:27, ncol = 4)
doPlots(df, fun = plotbxp, ii = 28:35, ncol = 4)
doPlots(df, fun = plotbxp, ii = 36:43, ncol = 4)
```

The plots below illsutrates the distribution of attributes for each meter state:

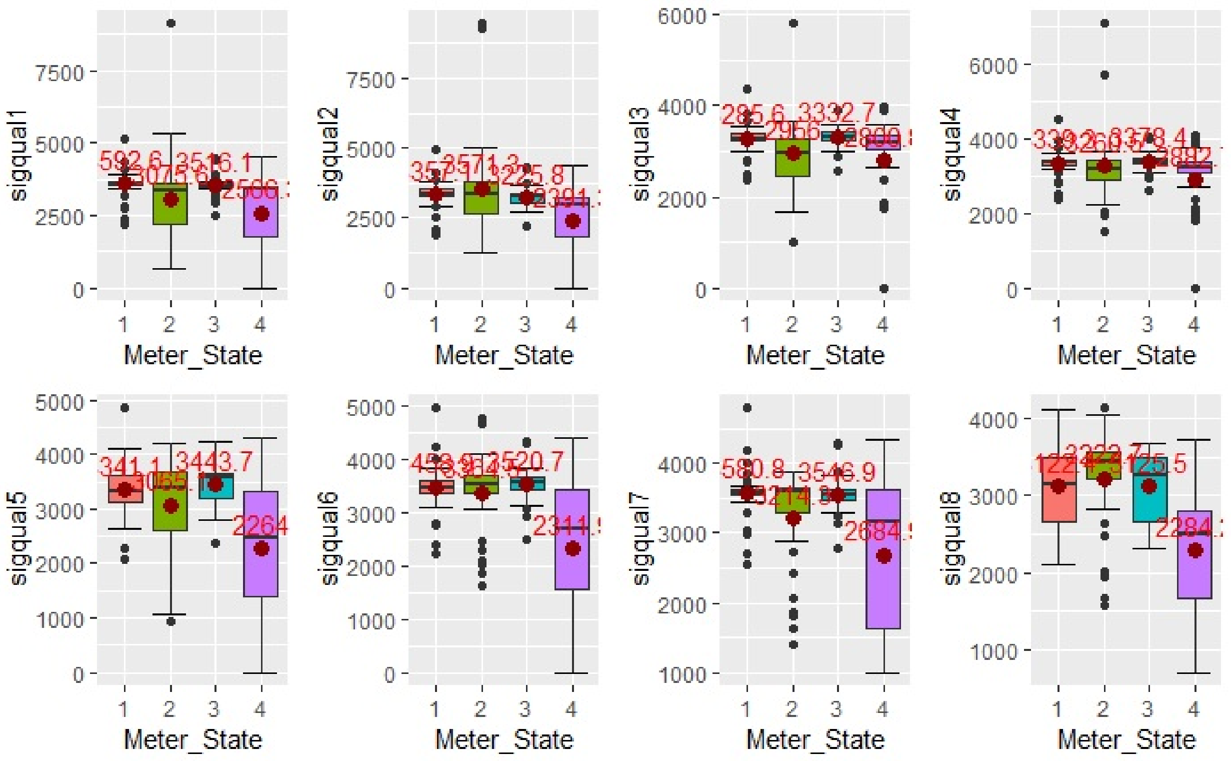


Figure 4: Boxplot of signal quality distribution by meter state

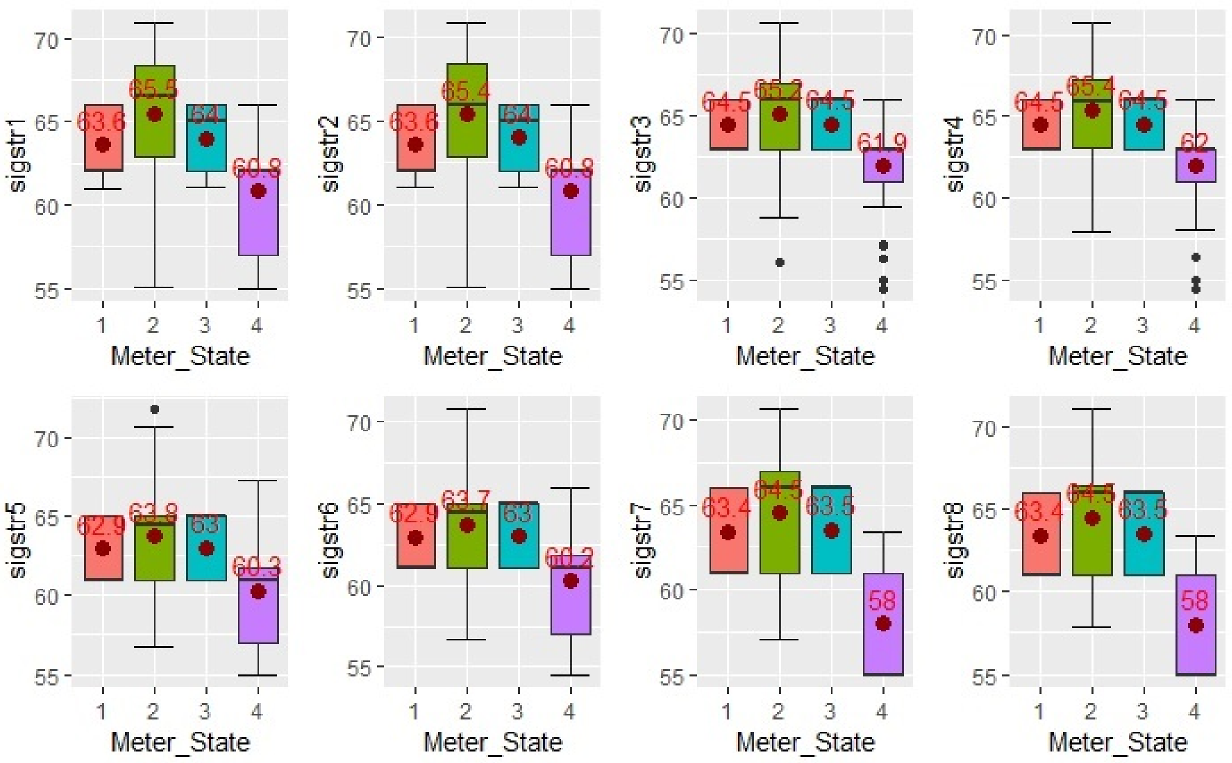


Figure 5: Boxplot of Signal strength distribution by meter state

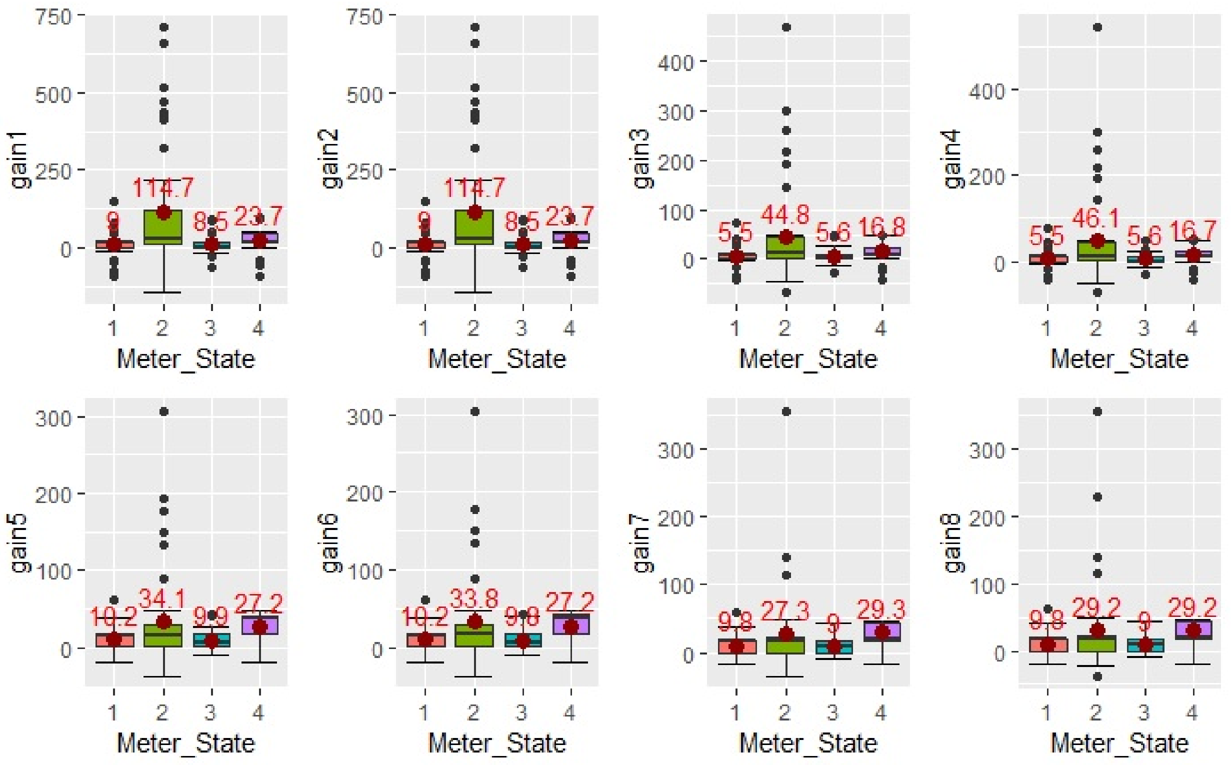


Figure 6: Boxplot of gain distribution by meter state

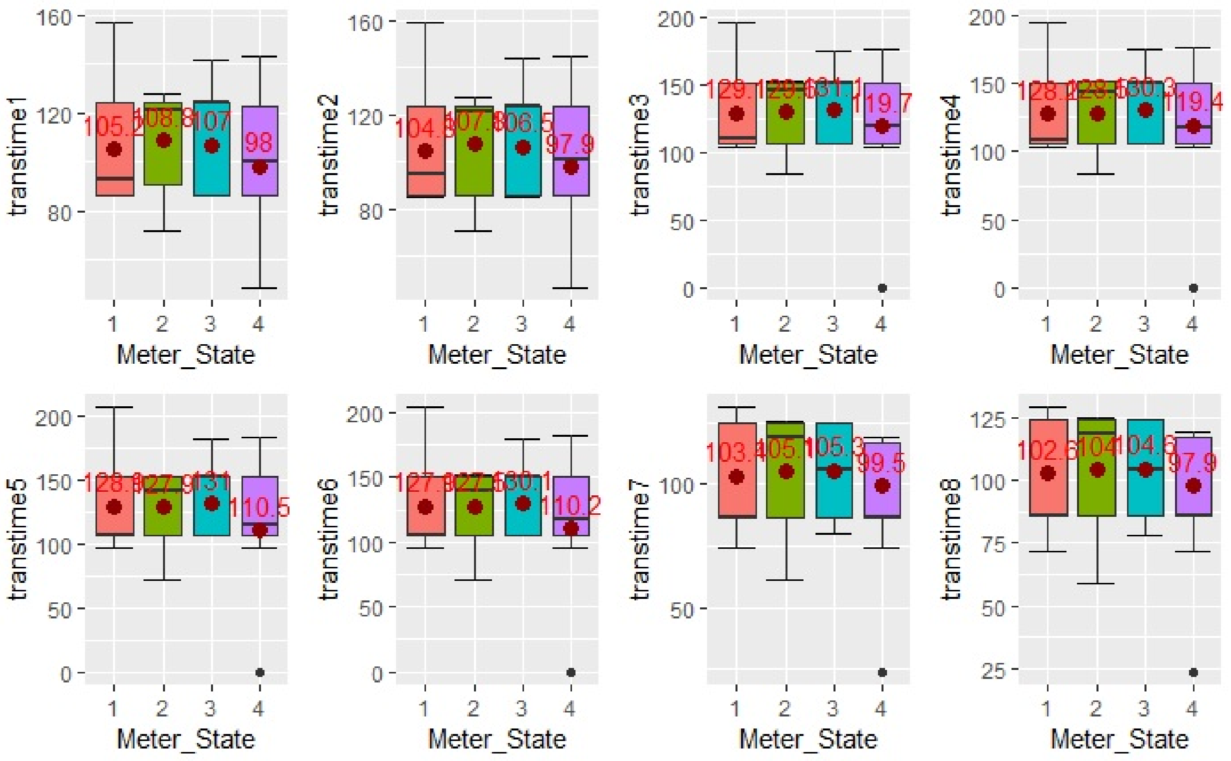


Figure 7: Boxplot of transit time by signal strength

From the boxplots of the predictive attributes, it can be seen that there is significant overlap between class 1 and 3 for all signal strengths (further plots have been omitted for clarity) This pattern was repeated across each attribute family, with the exception of transit time which showed a uniform distribution for all classes. Also highlighted by the dotplots were the presence of several high outlying values in the signal gain of the waxing meter state. This could potentially be real values, from when the meter is in a clean state and there is not wax layer to dampen the amplitude of the waves passed between each transmission/receiving point. As a result they have not been excluded in the training set, as in practice the meter will have to generalise across many operating conditions.

2.4 Introduction of noise to the dataset

The experimental work upon which the dataset was based [3] was conducted within a lab environment under experimental conditions. To simulate an industrial flowmeter, which will be subject to external noise and have to generalise over all working environments, normally distributed noise was introduced to 8% of instances for all the predictive attributes in the dataset using the function shown below:

```
noisydata <- df

nnoise <- function(data_in, i, j){
#set the seed to 0
 set.seed(0)

#create a copy of the dataframe
  noisydata <- data_in

#Choose % of instances to corrupt from a binomial distribution
 corrupt <- rbinom(nrow(noisydata),1, j)

#convert to boolean logic
 corrupt <- as.logical(corrupt)

#introduce normally distributed noise for each of the corrupted attributes
 noise <- rnorm(corrupt, median(noisydata[[i]]), sd(noisydata[[i]]))
 noisydata[[i]][corrupt] <- as.integer(noise[corrupt])
 return(noisydata)
}

#Create a loop that introduces noise to all the predictive attributes of the dataset
for(i in 1:43) {
noisydata <- nnoise(noisydata, i, 0.08)
}
```

The following four graphs show the distribution of four of the ”noisified” attributes compared to the underlying dataset to check that the noise introduced has not been excessive.

```
df <- noisydata
```

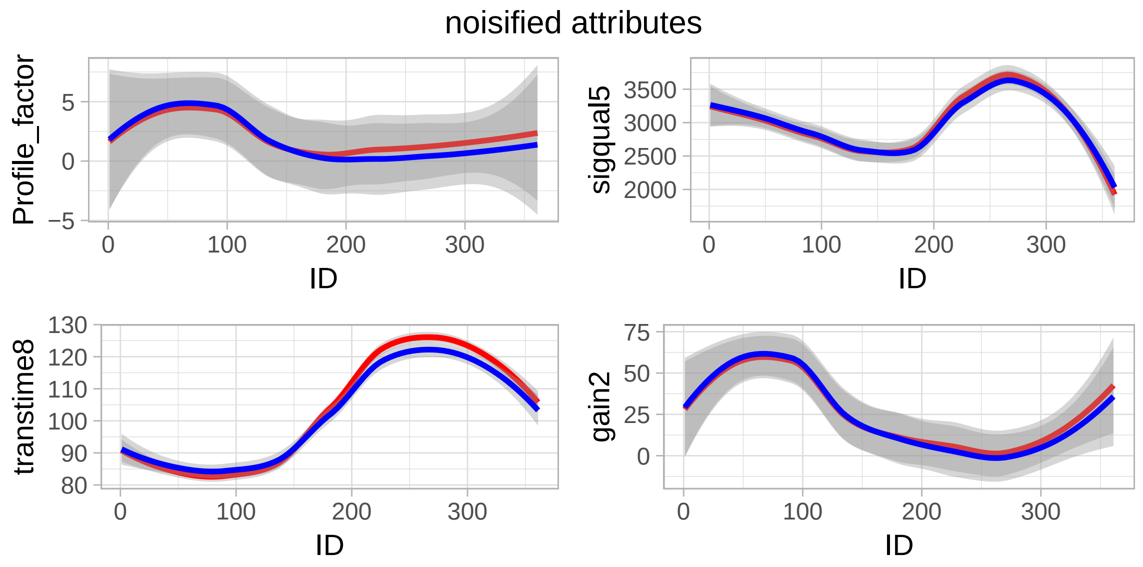


Figure 8: Noisified attributes

2.5 Principal component analysis

Principal component analysis is a technique used to capture the ”essence” of a dataset. It achieves this by summarising a high dimensionality data set in to a lower order, by successively fitting vectors through a *p*-dimensional space to explain the most variance in the dataset, where p is the number of attributes. In the context of this experiment, this will filter out variables that are too noisy or do not have a large contribution to the predictive accuracy.

The analysis was carried out using the in built ”prcomp” function in R.

```r
pca.flowmeters <- prcomp(df[,-44], center = TRUE, scale = TRUE)
View(pca.flowmeters$rotation)
```

```r
#calculate standard deviation of principal components
std_dev <- pca.flowmeters$sdev
#calculate variance
pr_var <- std_dev^2

#calculate cumulative proportion of variance explained by each principal component
prop_var_ex <- as.data.frame(pr_var/sum(pr_var))

#add row id as column to data frame, change to first row and rename variables
prop_var_ex$Principal_component <- seq.int(nrow(prop_var_ex))
prop_var_ex <- prop_var_ex[c(2,1)]
names(prop_var_ex)[2] <- "Proportion_of_variance_explained"

#Create Scree plot of proportion of variance explained against the number of principle components
p <- ggplot(data = prop_var_ex, aes(Principal_component, Proportion_of_variance_explained)) +
  geom_jitter() +
  ggtitle("Scree plot of variance explained by successive principle components") +
  theme(plot.title = element_text(hjust = 0.5))
p
```

As can be seen from the scree plot, the variance explained by each principle component is decreases considerably after 10 principle components (a summary of the total variance explained is given in Appendix section 6.3. The table below summarises what the top 4 principle components are comprised of based on their rotation coefficients (the first 4 principle components are available in appendix 6.2:

Principle Component	Top 5 attribute loadings	Theme of principle component
PC1	transtime1, sigqual5, transtime2, sigstr6, transtime3	Transit time across all four paths
PC2	sigqual7, FV4, sigqual8, FV2, sigstr7	Signal strength and quality
PC3	FV3, FV1, FV2, vsnd1, sigqual1	Fluid velocity
PC4	gain3, gain4, gain5, gain6, gain8	Signal gain

Table 1: Breakdown of principle component compositions

From analysis of the principal component rotations in the appendices section 6.2 it can be seen that principal component 1 is highly distributed across the ultrasonic attributes, though no one attribute or family of attributes appears to be predominate.

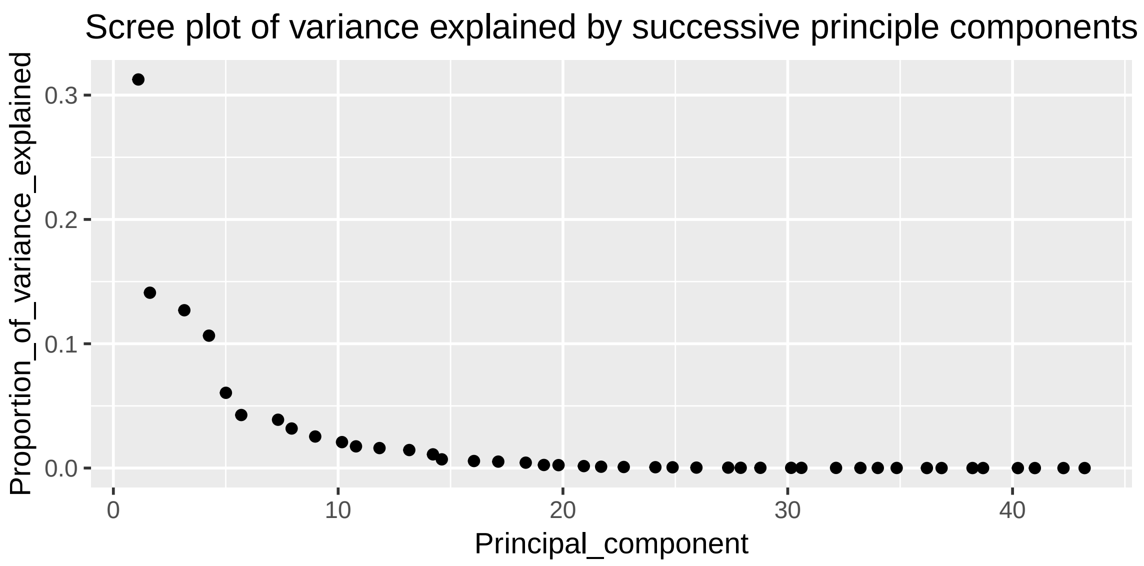


Figure 9: Scree plot of variance explained by principal components

3 Supervised learning experiments

3.1 Sampling Method

A 70% balanced split based on the Meter State was used between the testing a training sets which provided 255 observations for training and 106 for testing. This split was chosen as there were four class attributes with an imbalance which made the waxing attribute around half as probable to occur as the other three.

```
#set seed to guarantee repeatibility of split
set.seed(0)

#coerce Meter_state to factor
df$Meter_State <- factor(df$Meter_State)

#create 70% stratified split of training data
inTrain <- createDataPartition(y = df$Meter_State,
                               p = 0.7,
                               list = FALSE)

#assign instances in the dataframe to training dataset
train_nopca <- df[inTrain, ]

#assign remaining instances to test set
test_nopca <- df[-inTrain, ]
```

The training / testing split was then repeated in a similar method for the PCA pre processed data with a specific number of principle components selected for each experiment iteration.

```
#create a data frame of the PCA scores
train.data <- data.frame(pca.flowmeters$x)

#select only the first 15 principle components (this value was varied for different experiments)
```

```r
train.data <- train.data[,1:20]

#coerce Meter_state attribute to a factor
train.data$Meter_State <- factor(df$Meter_State)


#set seed to guarantee repeatibility of split
set.seed(0)

#create 70% stratified split of training data
inTrain_PCA <- createDataPartition(y = train.data$Meter_State,
                                   p = 0.7,
                                   list = FALSE)

#assign instances in the dataframe to training dataset
train_pca <- train.data[inTrain_PCA, ]

#assign remaining instances to test set
test_pca <- train.data[-inTrain_PCA, ]
```

In order to prevent testing the model on the same dataset it was trained upon, a bootstrap resampling method was used. Due to the bootstrap method sampling with replacement, it provides estimates which are optimistic of a classifiers accuracy. This was felt to be an acceptable trade off in the interest of ensuring that there was a representative number of each of the four class attributes in the training and test set for each repitition of the training.
The control paramters were set using the following code

```r
#set model control paramters
trctrl <- trainControl(method = "boot", number = 3, verboseIter = FALSE)
```

3.2 Algorithm

The code for producing each model is displayed below for models with and without PCA with the same common resampling parameters set in section 3.1. Two distinctly different algorithms were chosen to explore the effect of PCA on each;

- Linear support vector machines (SVMLinear) that constructs hyperplanes that seperate the clusters of data. The application of PCA will be interesting as the dimension reduction assoicciated with PCA is seemingly at odds with the high dimensionality required for SVMLinear.
- C5.0, which uses the concept of information entropy to construct a boosted decision tree. Pre processing with PCA will result in a smaller tree, though sacrifice clarity on the splitting attribute, which should mitigate the well known problem of overfitting with decision trees.

3.3 Performance metrics

The classification was performed using predictive accuracy to optimise the tuning parameters and evaluate the training / testing performance of each classifier. Using Cohens Kappa statistic could improve classification accuracy for highly imbalanced datasets, but the imbalance is not considered sufficiently high for any advantage to be conferred by it's use. In conjunction with the classifier accuracy on the training and testing sets, a confusion matrix was constructed to highlight which instances were correctly / incorrectly classified by the model.
The time to train each algorithm was measured (though this has an uncertainty attached to it due to variable RAM availability) to highlight the ability of the algorithm to be scaled. PCA significantly reduces the size of the data frame used in training; whilst this dataset is relatively small, many flowmeters operate continuously in real time generating a large amount of signal data, where storage / processor availability may place restrictions on the model.

3.4 Tuning

For both SVMLinear and C5.0 algorithms, the number of principle components was altered to find the optimum value for each. From visual inspection of the scree plot in figure 2.5, this would be seem to be around 10, but in practice the values which proved to sufficiently capture the complexity of the dataset were between 14 and 16 without overfitting.
For the C5.0 algorithm, three tuning parameters are available:

- trials - an integer variable representing the number of boosting iterations to test, the a vector of 1 to 20 rounds of boosting was used as this would test the algorithm without incurring excessive processing time
- winnow - a boolean value representing whether to conduct feature selection, this was set to false for all experiments as it will automatically exclude noisy attributes to avoid overfitting, and bias the results
- tree rules - whether to use a tree based algorithm or rules based - Only tree based models were tested as rule based classifiers offer a higher degree of flexibility when dealing with overfitting and noise.

For SVMLinear, the only paramater that requires tuning is the cost parameter C. This optimises the distance between the seperating hyperplanes to minimise the classification error on the test set.

3.4.1 C5.0 model without PCA

```r
#initialise time counter
start_time_C5_nopca <- Sys.time()

#set seed to ensure repeatability of experiment
set.seed(0)
```

```r
#train classifier with parameters described below
C5_model_nopca <- train(Meter_State ~ .,
                    data = train_nopca,
                    method = "C5.0",
                    trControl = trctrl,
                    tuneGrid = expand.grid(.trials = c(1:20),
                                           .winnow = FALSE,
                                           .model = "tree"))

#stop timer
end_time_C5_nopca <- Sys.time()
```

3.4.2 C5.0 model with PCA

```r
#initialise time counter
start_time_C5_pca <- Sys.time()

#set seed to ensure repeatability of experiment
set.seed(0)

#train classifier with parameters described below
C5_PCA_model <- train(Meter_State ~ .,
                    data = train_pca,
                    method = "C5.0",
                    trControl = trctrl,
                    tuneGrid = expand.grid(.trials = c(1:20),
                                           .winnow = FALSE,
                                           .model = "tree"))
#stop timer
end_time_C5_pca <- Sys.time()
```

3.4.3 SVMLinear model without PCA

```r
#initialise time counter
start_time_SVM_nopca <- Sys.time()

#set seed to ensure repeatability of experiment
set.seed(0)

#train classifier with parameters described below
SVM_model_nopca <- train(Meter_State ~ .,
                    data = train_nopca,
                    method = "svmLinear",
                    tuneGrid = expand.grid( C = c(1:15)),
                    trControl = trctrl)
#stop timer
end_time_SVM_nopca <- Sys.time()
```

3.4.4 SVMLinear model with PCA

```
#initialise time counter
start_time_SVM_pca <- Sys.time()

#set seed to ensure repeatability of experiment
set.seed(0)

#train classifier with parameters described below
SVM_PCA_model <- train(Meter_State ~ .,
                    data = train_pca,
                    method = "svmLinear",
                    tuneGrid = expand.grid( C = c(1:15)),
                    trControl = trctrl)
#stop timer
end_time_SVM_pca <- Sys.time()
```

3.5 Evaluation of results

3.5.1 C5.0 Model without PCA results

```
#predict values using C5.0 model
pred_C5.test <- predict(C5_model_nopca, test_nopca[,-ncol(test_nopca)], type = "raw")
pred_C5.train <- predict(C5_model_nopca, train_nopca[,-ncol(train_nopca)], type = "raw")

#evaluate classifier on training set
table(pred_C5.train, train_nopca$Meter_State)

##
## pred_C5.train  1  2  3  4
##             1 72  0  0  0
##             2  0 33  0  0
##             3  1  0 77  0
##             4  1  0  0 71

cat("C5.0 model, without PCA, training set Accuracy; ",
    mean(pred_C5.train==train_nopca$Meter_State)*100,"%")

## C5.0 model, without PCA, training set Accuracy;  99.21569 %

#evaluate classifier on test set
table(pred_C5.test, test_nopca$Meter_State)

##
## pred_C5.test  1  2  3  4
##            1 28  0  5  0
##            2  0 13  0  0
##            3  3  0 27  2
##            4  0  0  0 28

cat("C5.0 model, without PCA, test set Accuracy; ",
    mean(pred_C5.test==test_nopca$Meter_State)*100,"%", "\n", "CPU time =",
    end_time_C5_nopca - start_time_C5_nopca, "s")

## C5.0 model, without PCA, test set Accuracy;  90.56604 %
##  CPU time = 2.345549 s
```

3.5.2 C5.0 model with PCA results (20 principle components)

```
#predict values using C5.0 model
pred_C5_PCA.test <- predict(C5_PCA_model, test_pca[,-ncol(test_pca)], type = "raw")
pred_C5_PCA.train <- predict(C5_PCA_model, train_pca[,-ncol(train_pca)], type = "raw")

#evaluate classifier on training set
table(pred_C5_PCA.train, train_pca$Meter_State)

##
## pred_C5_PCA.train  1  2  3  4
##                 1 74  0  0  0
##                 2  0 33  0  0
##                 3  0  0 77  0
##                 4  0  0  0 71

cat("C5.0 model, with PCA, training set Accuracy; ",
    mean(pred_C5_PCA.train==train_pca$Meter_State)*100,"%")

## C5.0 model, with PCA, training set Accuracy;  100 %

#evaluate classifier on test set
table(pred_C5_PCA.test, test_pca$Meter_State)

##
## pred_C5_PCA.test  1  2  3  4
##                1 23  0  3  2
##                2  2 12  0  0
##                3  5  0 29  3
##                4  1  1  0 25

cat("C5.0 model, with PCA, test set Accuracy; ",
    mean(pred_C5_PCA.test==test_pca$Meter_State)*100,"%", "\n",
    "CPU time =", end_time_C5_pca - start_time_C5_pca, "s")

## C5.0 model, with PCA, test set Accuracy;  83.96226 %
##  CPU time = 1.406267 s
```

3.5.3 SVMLinear model without PCA results

```
#predict values using SVM model
pred_SVM.test <- predict(SVM_model_nopca, test_nopca[,-ncol(test_nopca)], type = "raw")
pred_SVM.train <- predict(SVM_model_nopca, train_nopca[,-ncol(train_nopca)], type = "raw")

#evaluate classifier on training set
table(pred_SVM.train, train_nopca$Meter_State)

##
## pred_SVM.train  1  2  3  4
##              1 62  1  6  0
##              2  0 32  0  0
##              3  9  0 71  1
##              4  3  0  0 70

cat("SVM model without PCA training set Accuracy; ",
    mean(pred_SVM.train==train_nopca$Meter_State)*100,"%")

## SVM model without PCA training set Accuracy;  92.15686 %

#evaluate classifier on test set
table(pred_SVM.test, test_nopca$Meter_State)

##
## pred_SVM.test  1  2  3  4
##             1 27  2  2  1
##             2  0  7  0  0
##             3  3  2 29  1
##             4  1  2  1 28

cat("SVM model without PCA test set Accuracy; ",
    mean(pred_SVM.test==test_nopca$Meter_State)*100,"%", "\n",
    "CPU time =", end_time_SVM_nopca - start_time_SVM_nopca, "seconds")

## SVM model without PCA test set Accuracy;  85.84906 %
##  CPU time = 2.092405 seconds
```

3.5.4 SVMLinear model with PCA results (20 principle components)

```
#predict values using SVM model
pred_SVM_PCA.test <- predict(SVM_PCA_model, test_pca[,-ncol(test_pca)], type = "raw")
pred_SVM_PCA.train <- predict(SVM_PCA_model, train_pca[,-ncol(train_pca)], type = "raw")

#evaluate classifier on training set
table(pred_SVM_PCA.train, train_pca$Meter_State)

##
## pred_SVM_PCA.train  1  2  3  4
##                  1 68  0 23  1
##                  2  0 33  0  0
##                  3  2  0 52  1
##                  4  4  0  2 69

cat("SVM model with PCA training set Accuracy; ",
    mean(pred_SVM_PCA.train==train_pca$Meter_State)*100,"%")

## SVM model with PCA training set Accuracy;  87.05882 %

#evaluate classifier on test set
table(pred_SVM_PCA.test, test_pca$Meter_State)

##
## pred_SVM_PCA.test  1  2  3  4
##                 1 30  1 11  3
##                 2  0  9  0  0
##                 3  0  1 21  1
##                 4  1  2  0 26

cat("SVM model with PCA test set Accuracy; ",
    mean(pred_SVM_PCA.test==test_pca$Meter_State)*100,"%", "\n",
    "CPU time =", end_time_SVM_pca - start_time_SVM_pca, "seconds")

## SVM model with PCA test set Accuracy;  81.13208 %
##  CPU time = 0.9414849 seconds
```

3.6 Comparison of results

The table below summarises the outcome of successive experiments carried out with no principle component analysis and then successive runs of 10, 20, 30 and 40 principle components using the algorithms described in section 3.2

Algorithm	**Accuracy (%)**		**Accuracy loss on testing**	**CPU Time (s)**
	Train	**Test**		
C5.0 no noise, no PCA	99.6	90.6	9.0	2.6
C5.0 without PCA	99.2	90.6	8.6	2.6
C5.0 with PCA (40 PC's)	100	82.1	17.9	2.9
C5.0 with PCA (30 PC's)	100	85.8	14.2	2.2
C5.0 with PCA (20 PC's)	100	84.0	16.0	1.7
C5.0 with PCA (10 PC's)	99.6	72.6	27	1.2
SVMLinear no noise, no PCA	92.2	89.6	2.6	2.6
SVMLinear without PCA	92.1	85.5	6.6	1.6
SVMLinear with PCA (40 PC's)	93.7	81.1	12.6	1.5
SVMLinear with PCA (30 PC's)	93.3	82.1	11.2	1.5
SVMLinear with PCA (20 PC's)	87.1	81.1	6.0	1.4
SVMLinear with PCA (10 PC's)	85.5	84.0	1.5	1.0

Table 2: Results summary table

The table below illustrates the relative effectiveness of each of the four algorithms with and without PCA with a 95% confidence interval.

```r
# collect resamples
results <- resamples(list(C5.0_noPCA=C5_model_nopca,
                          C5.0_PCA=C5_PCA_model,
                          SVM_noPCA=SVM_model_nopca,
                          SVM_PCA=SVM_PCA_model))

#Plot resamples as a dotplot with 95% confidence interval
scales <- list(x=list(relation="free"), y=list(relation= "free"))
dotplot(results, scales=scales, conf.level = 0.95)
```

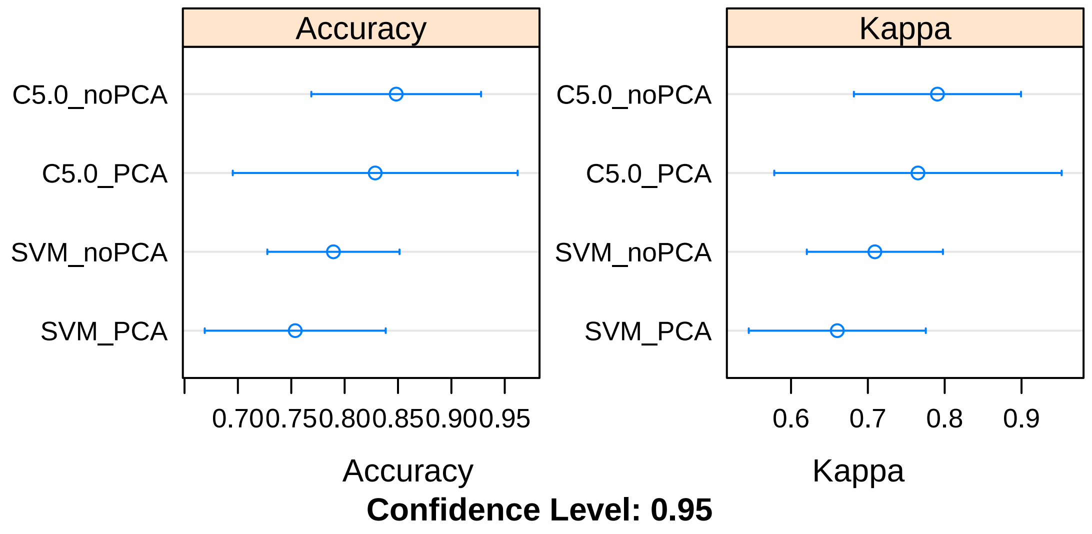


Figure 10: Comparison of results

4 Discussion

From the experiments summarised in table 2 and figure 10 it can be seen that C5.0 outperforms SVMLinear in terms of classification accuracy, but at the cost of a slightly higher processing time. This is owing to SVMLinear making the assumption of linear seperability for all four classes, which is not true between all classes, C5.0 on the other hand makes few assumptions about the dataset and therefore has a higher accuracy. In relation to the prediction of a meter state, having non linearly seperable data would confer that there is some kind of transitional region where, when looking at the attributes in the datsset, the fluid could belong to multiple meter states. In the context of robustness to noise, the c5.0 algorithm appears to learn the same tree with and without moderate amounts of noise, whereas SVMLinear is creating hyperplanes that are misclassifying instances, this is reflected in the test set accuracy for each.

Both algorithms had difficulty discriminating between the gas injection and healthy meter states, which is indicated by the significant overlap of the boxplots as seen in figure 5. This also supports the conclusion of the dataset not being linearly seperable. A potential cause of this is that the volume of gas injected is relatively small compared to the bulk volume of the fluid, therefore is not contributing significantly to the ultrasonic measurement. Potentially a new attribute that more highly correlates with gas injection, such as density, could be added to the dataset to improve the classification accuracy between these variables.

The introduction of PCA resulted in a drop in predictive accuracy and a wider confidence interval as shown in figure 10 for both C5.0 and SVMLinear. This contradicts the hypothesis that PCA would help correct the problem of overfitting noise, as the model seems to be gaining more predictive accuracy, even with the noisified attributes being added. This was seen initially from the rotations of PC1 (see section 6.2) as each of the attributes relating to the ultrasonic measurements appears to contribute evenly to the principal component. SVMLinear was much more resistant to overfitting than C5.0, as it not only benefits from PCA but also in allowing for a margin of error through the tuning of the C parameter.

The decision tree produced by the C5.0 algorithm is too large to be reproduced in this report; however, upon inspection in R studio, the top splitting attribute is signal strength 1 whereas with the preprocessed data it is principal component 4 which is predominantly comprised of the signal gains. This is surprising as signal strength 1 does not have a high rotation coefficient in the first principal component. The mismatch between the PCA and the decision tree again supports the conclusion that the maximum discriminating power for the meter state is not being found in the dimension with the most variation and as a result PCA is not proving effective for pre processing

5 References

[1] Ja Hyon Ku. "Chapter 45 - J.W. Strutt, third Baron Rayleigh, The theory of sound, first edition (1877–1878)". In: *Landmark Writings in Western Mathematics 1640-1940*. Ed. by I. Grattan-Guinness et al. Amsterdam: Elsevier Science, 2005, pp. 588 –599. ISBN: 978-0-444-50871-3. DOI: https://doi.org/10.1016/B978-044450871-3/50126-1. URL: http://www.sciencedirect.com/science/article/pii/B9780444508713501261.

[2] fluidic ltd. *Ultrasonic flowmeter principle of operation Ultrasonic flowmeter principle of operation*. 1999. URL: https://fluidic-ltd.co.uk/product/micronics-u3000/ (visited on 03/29/2019).

[3] Kojo Sarfo Gyamfi et al. "Linear dimensionality reduction for classification via a sequential Bayes error minimisation with an application to flow meter diagnostics". In: *Expert Systems with Applications* 91 (2018), pp. 252 –262. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2017.09.010. URL: http://www.sciencedirect.com/science/article/pii/S0957417417306140.

[4] Emerson. *Four path pipe configuration Four path pipe configuration*. 1999. URL: https://www.emerson.com/en-gb/automation/measurement-instrumentation/flow-measurement/ultrasonic (visited on 03/30/2019).

6 Appendix

6.1 Dataset attribute description

1. `Profile_factor`: Numerical value relating to the velocity profile of the fluid
2. `Symmetry`: Numerical value relating to the symmetry of the fluids velocity profile
3. `Crossflow`: Flow perpendicular to main direction of flow
4. `FV[x]`: Flow velocity at each of the four sonic paths
5. `vsnd[x]`: Speed of sound in each of the four sonic paths
6. `sigstr[x]`: Signal strength at both transmitter / receiver of each of the four sonic paths
7. `sigqual[x]`: Signal quality both transmitter / receiver of each of the four sonic paths
8. `Gain[x]`: Signal gain at both transmitter / receiver of each of the four sonic paths
9. `transtime[x]`: Time at both transmitter / receiver of each of the four sonic paths
10. `Meter_state`: Fluid operational state

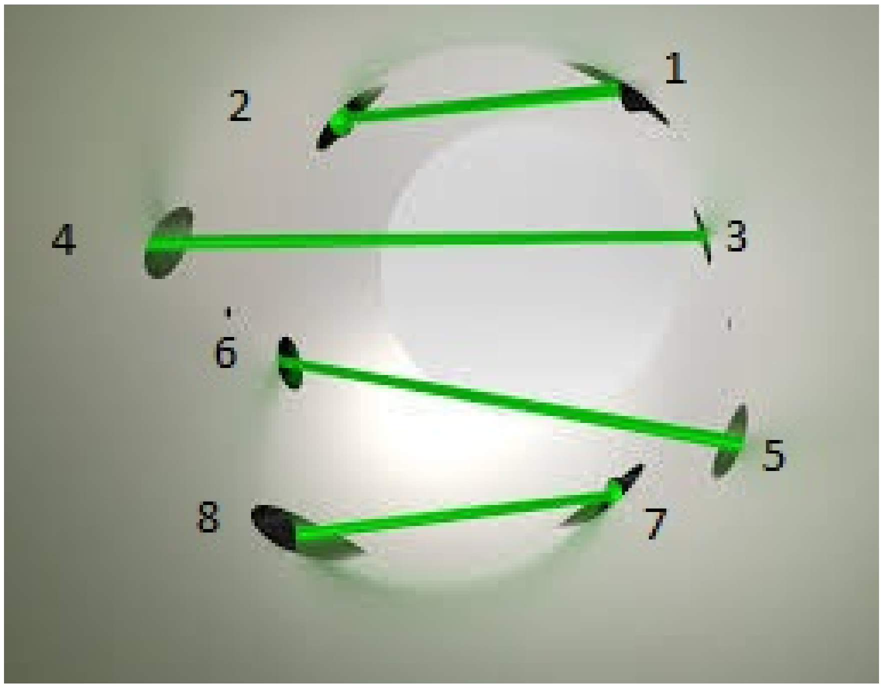
path.jpg 

Figure 11: Four path pipe configuration [4]

6.2 Principle component analysis rotations

```
##                       PC1         PC2         PC3          PC4
## Profile_factor  0.003275155 -0.03370711  0.16071402  0.0681693719
## Symmetry        0.038097144 -0.04353676  0.06927055 -0.0090216298
## Crossflow       0.108437411  0.03836605  0.20265517 -0.0915447134
## FV1             0.001902622 -0.27359835  0.16102193 -0.2003327525
## FV2            -0.031979029 -0.29481148  0.17448398 -0.1649821660
## FV3             0.016003647 -0.24180075  0.22582882 -0.1700858143
## FV4            -0.068782837 -0.33595301  0.02519271 -0.0404215446
## vsnd1          -0.122076941 -0.04954682 -0.14585444  0.2089710212
## vsnd2           0.144578795 -0.03548878  0.18109423 -0.1608157059
## vsnd3           0.140352052  0.05691760  0.15429306 -0.0689428659
## vsnd4           0.051512214  0.10214939  0.13347101 -0.0368367485
## sigstr1         0.210549065 -0.14162747 -0.08178001  0.1575130117
## sigstr2         0.210829430 -0.13952295 -0.08534534  0.1512232724
## sigstr3         0.210089326 -0.15887197 -0.02809813  0.1040399951
## sigstr4         0.206802778 -0.16221198 -0.01902187  0.1180414998
## sigstr5         0.204809542 -0.12357950 -0.06325243  0.1715656660
## sigstr6         0.211656398 -0.12351063 -0.06423527  0.1587865564
## sigstr7         0.170359353 -0.18821279 -0.13095190  0.1823393469
## sigstr8         0.169972888 -0.18740521 -0.12925015  0.1852268280
## sigqual1        0.168315094 -0.09648454  0.13597499 -0.2116656340
## sigqual2        0.157476291 -0.07426679  0.18748368 -0.1135731729
## sigqual3        0.135085299 -0.09521264  0.11768914 -0.2298729725
## sigqual4        0.132973274 -0.07915422  0.17116491 -0.1712631055
## sigqual5        0.223469086 -0.01734019  0.03672225 -0.0974546813
## sigqual6        0.204303007 -0.05601468  0.09428520 -0.0879947794
## sigqual7        0.075473065 -0.29907692 -0.15481858  0.0001338682
## sigqual8        0.164024613 -0.18423420 -0.13470369  0.1295121333
## gain1          -0.020433575 -0.07268752  0.15248959  0.1621596897
## gain2          -0.020433575 -0.07268752  0.15248959  0.1621596897
## gain3          -0.056428702 -0.05489754  0.25349001  0.2837514370
## gain4          -0.052403605 -0.05757977  0.24840099  0.2814172441
## gain5          -0.119458406 -0.04166164  0.24496719  0.2628673627
## gain6          -0.120790307 -0.04127976  0.24324513  0.2612870637
## gain7          -0.079964569  0.04372233  0.31177884  0.1586657806
## gain8          -0.076911654  0.03835262  0.30742718  0.1720896019
## transtime1      0.220076605  0.19622401  0.06880558  0.0361713401
## transtime2      0.216529853  0.20256404  0.06672098  0.0340951409
## transtime3      0.209208528  0.17251854  0.07769340 -0.0191768244
## transtime4      0.208182577  0.17589680  0.07915727 -0.0120740234
## transtime5      0.192274380  0.18990655  0.07663482  0.0263375502
## transtime6      0.190349626  0.19415132  0.07393602  0.0306298624
## transtime7      0.206638445  0.17569470 -0.02829777  0.1319510135
## transtime8      0.206549260  0.16519275 -0.03707990  0.1353387809
```

6.3 Cumulative proportion of variance

```
#display summary of PCA
summary(pca.flowmeters)

## Importance of components:
##                           PC1    PC2    PC3    PC4     PC5     PC6     PC7
## Standard deviation     3.6662 2.4627 2.3364 2.1400 1.61337 1.35462 1.29269
## Proportion of Variance 0.3126 0.1411 0.1270 0.1065 0.06053 0.04267 0.03886
## Cumulative Proportion  0.3126 0.4536 0.5806 0.6871 0.74761 0.79029 0.82915
##                            PC8     PC9    PC10    PC11    PC12    PC13
## Standard deviation     1.16958 1.04454 0.94717 0.86654 0.83182 0.78939
## Proportion of Variance 0.03181 0.02537 0.02086 0.01746 0.01609 0.01449
## Cumulative Proportion  0.86096 0.88633 0.90720 0.92466 0.94075 0.95524
##                           PC14    PC15    PC16    PC17    PC18    PC19
## Standard deviation     0.68852 0.54947 0.49354 0.47200 0.43169 0.32655
## Proportion of Variance 0.01102 0.00702 0.00566 0.00518 0.00433 0.00248
## Cumulative Proportion  0.96627 0.97329 0.97895 0.98413 0.98847 0.99095
##                           PC20    PC21    PC22    PC23    PC24    PC25
## Standard deviation     0.31851 0.26059 0.21462 0.19183 0.17098 0.16210
## Proportion of Variance 0.00236 0.00158 0.00107 0.00086 0.00068 0.00061
## Cumulative Proportion  0.99331 0.99489 0.99596 0.99681 0.99749 0.99810
##                           PC26    PC27    PC28    PC29    PC30    PC31
## Standard deviation     0.12452 0.11983 0.09452 0.09188 0.08506 0.07729
## Proportion of Variance 0.00036 0.00033 0.00021 0.00020 0.00017 0.00014
## Cumulative Proportion  0.99846 0.99880 0.99901 0.99920 0.99937 0.99951
##                           PC32    PC33    PC34    PC35    PC36    PC37
## Standard deviation     0.07656 0.06527 0.06347 0.05680 0.03563 0.02842
## Proportion of Variance 0.00014 0.00010 0.00009 0.00008 0.00003 0.00002
## Cumulative Proportion  0.99965 0.99974 0.99984 0.99991 0.99994 0.99996
##                           PC38    PC39    PC40     PC41     PC42      PC43
## Standard deviation     0.02721 0.02251 0.01652 0.007981 0.007088 7.134e-16
## Proportion of Variance 0.00002 0.00001 0.00001 0.000000 0.000000 0.000e+00
## Cumulative Proportion  0.99998 0.99999 1.00000 1.000000 1.000000 1.000e+00
```

6.4 Full source code

```
#set working directory

setwd("F:/University/CMM535 Data science development/Flowmeters")
getwd()

#Load in relevant libraries and check installation status
Sys.setenv(JAVA_HOME='C:\\Program Files\\Java\\jre7')
load.libraries <- c('tidyverse', 'gridExtra', 'corrplot', 'caret', 'GGally', 'ggplot2', 'dplyr', 'RWeka
install.lib <- load.libraries[!load.libraries %in% installed.packages()]
for(libs in install.lib) install.packages(libs, dependences = TRUE)
sapply(load.libraries, require, character = TRUE)


#read in data from file

dfC <- read.delim("Meter C", header = FALSE)
dfD <- read.delim("Meter D", header = FALSE)



df <- rbind(dfC, dfD)
rm(dfC, dfD)


#rename filenames

names(df) <- c("Profile_factor", "Symmetry", "Crossflow",
               "FV1", "FV2", "FV3","FV4",
               "vsnd1", "vsnd2", "vsnd3", "vsnd4",
               "sigstr1", "sigstr2", "sigstr3","sigstr4","sigstr5", "sigstr6", "sigstr7","sigstr8",
               "sigqual1", "sigqual2", "sigqual3", "sigqual4","sigqual5", "sigqual6", "sigqual7", "sigq
               "gain1", "gain2", "gain3", "gain4","gain5", "gain6", "gain7", "gain8",
               "transtime1", "transtime2", "transtime3", "transtime4","transtime5", "transtime6", "trans
               "Meter_State")




str(df)
colSums(sapply(df, is.na))




#Plot correlation matrix of variables and

#calculate Pearsons correlation coefficients for data frame, marking uncorrelated variables with an X a
res1 <- cor.mtest(df, conf.level = 0.95)

#Generate correlation matrix for data frame
correlation_matrix <- corrplot(corr = cor(df),
                               p.mat = res1$p,
                               method = "color",
                               tl.col = "black",
                               tl.srt = 45,
                               type = "lower",
                               title = "Correlation matrix of flowmeter attributes, (uncorrelated attri
                               tl.cex = 0.7)


#Plot class attribute distribution

a <- ggplot(data = df, aes(x = Meter_State)) +
  geom_bar(stat = "count", fill = "green") +
  theme_light() +
  labs(title = "Plot of Meter state distribution")
a

#create boxplots of variables

plotbxp <- function(data_in, i, j = ncol(data_in)) {
  p <- ggplot(data_in, aes(x=data_in[[j]], y=data_in[[i]], fill = data_in[[j]])) +
    stat_boxplot(geom = 'errorbar') + geom_boxplot() +
    xlab(colnames(data_in)[j]) + ylab(colnames(data_in)[i]) +
    stat_summary(fun.y=mean, colour="darkred", geom="point", hape=18, size=3,show_guide = FALSE) +
    stat_summary(fun.y=mean, colour="red", geom="text", show_guide = FALSE, vjust=-0.7, aes( label=roun
    theme(legend.position = "none")


  return (p)
}

#display boxplots as grids

doPlots <- function(data_in, fun, ii, ncol=3) {
  pp <- list()
  for (i in ii) {
    p <- fun(data_in=data_in, i=i)
    pp <- c(pp, list(p))
  }
  do.call("grid.arrange", c(pp, ncol=ncol))
}


#plot reaction time of participant (tau), price elasticity (g) and power consumed (p) values in boxplot

doPlots(df, fun = plotbxp, ii = 1:3, ncol = 2)
doPlots(df, fun = plotbxp, ii = 4:7, ncol = 2)
doPlots(df, fun = plotbxp, ii = 8:11, ncol = 2)
doPlots(df, fun = plotbxp, ii = 12:19, ncol = 4)
doPlots(df, fun = plotbxp, ii = 20:27, ncol = 4)
doPlots(df, fun = plotbxp, ii = 28:35, ncol = 4)
doPlots(df, fun = plotbxp, ii = 36:43, ncol = 4)
```
```
```

```
#######################################Introduce noise to dataset######################################
#make a copy of dataframe
noisydata <- df




#Create function that creates a percentage of normal randomly distributed noise in a variable (indexed

nnoise <- function(data_in, i, j){
#set the seed to 0
 set.seed(0)

#create a copy of the dataframe
  noisydata <- data_in

#Choose % of instances to corrupt from a binomial distribution
 corrupt <- rbinom(nrow(noisydata),1, j)

#convert to boolean logic
 corrupt <- as.logical(corrupt)

#introduce normally distributed noise for each of the corrupted attributes
 noise <- rnorm(corrupt, median(noisydata[[i]]), sd(noisydata[[i]]))
 noisydata[[i]][corrupt] <- as.integer(noise[corrupt])
 return(noisydata)
}


#Create noise in all predictive variables

for(i in 1:43) {
noisydata <- nnoise(noisydata, i, 0.08)
}




#Create plots of baseline predictive attributes against "noisified" predictive attributes

noisydata$ID <- seq.int(nrow(noisydata))
df$ID <- seq.int(nrow(df))

b <- ggplot() +
   geom_smooth(data = df, aes(ID, Profile_factor), color = "red") +
   geom_smooth(data = noisydata, aes(ID, Profile_factor), color = "blue") +
   theme_light()
b

c <- ggplot() +
   geom_smooth(data = df, aes(ID, sigqual5), color = "red") +
   geom_smooth(data = noisydata, aes(ID, sigqual5), color = "blue") +
   theme_light()


d <- ggplot() +
   geom_smooth(data = df, aes(ID, transtime8), color = "red") +
   geom_smooth(data = noisydata, aes(ID, transtime8), color = "blue") +
   theme_light()

e <- ggplot() +
   geom_smooth(data = df, aes(ID, gain2), color = "red") +
   geom_smooth(data = noisydata, aes(ID, gain2), color = "blue") +
   theme_light()

grid.arrange(b,c,d,e, nrow = 2, ncol = 2, top = "noisified attributes")

noisydata$ID <- NULL
df$ID <- NULL





#Reassign noisified data back to dataframe label
 df <- noisydata




################################################Conduct PCA of dataset##################################




#conduct PCA on values in dataframe
pca.flowmeters <- prcomp(df[,-44], center = TRUE, scale = TRUE)
#View(pca.flowmeters$rotation)

#Visualisation of principal components is too messy
#ggbiplot(pca.flowmeters, groups = df$Meter_State, choice = c(1,2))






#display summary of PCA
summary(pca.flowmeters)




#calculate standard deviation of principal components
std_dev <- pca.flowmeters$sdev
#calculate variance
```

~~~~
pr_var <- std_dev^2

#calculate cumulative proportion of variance explained by each principal component
prop_var_ex <- as.data.frame(pr_var/sum(pr_var))

#add row id as column to data frame, change to first row and rename variables
prop_var_ex$Principal_component <- seq.int(nrow(prop_var_ex))
prop_var_ex <- prop_var_ex[c(2,1)]
names(prop_var_ex)[2] <- "Proportion_of_variance_explained"

#Create Scree plot of proportion of variance explained against the number of principle components
p <- ggplot(data = prop_var_ex, aes(Principal_component, Proportion_of_variance_explained)) +
  geom_jitter() +
  ggtitle("Scree plot of variance explained by successive principle components") +
  theme(plot.title = element_text(hjust = 0.5))
p


####################################C5.0 without PCA###################################################
#set model control parameters

#set model control paramters
trctrl <- trainControl(method = "boot", number = 3, verboseIter = FALSE)

#Coerce Meter_state to factor
df$Meter_State <- factor(df$Meter_State)
```

#create training and testing sets on whole data set with no noise and no PCA

set.seed(0)

#Divide data in to 70% training and 30% testing
inTrain <- createDataPartition(y = df$Meter_State,
                               p = 0.7,
                               list = FALSE)

train_nopca <- df[inTrain, ]
test_nopca <- df[-inTrain, ]




#create training and testing sets

#create data frame of pca scores

train.data <- data.frame(pca.flowmeters$x)

#take first 13 principle components

train.data <- train.data[,1:20]
~~~~

```r
#append meter_state data to principle component data

train.data$Meter_State <- factor(df$Meter_State)

set.seed(0)

#Divide data in to 70% training and 30% testing
inTrain_PCA <- createDataPartition(y = train.data$Meter_State,
                               p = 0.7,
                               list = FALSE)

train_pca <- train.data[inTrain_PCA, ]
test_pca <- train.data[-inTrain_PCA, ]




h <- ggplot(data = train_nopca, aes(x = Meter_State)) +
  geom_bar(stat = "count", fill = "green") +
  theme_light() +
  labs(title = "Training data Meter state distribution (nopca)")
h

i <- ggplot(data = test_nopca, aes(x = Meter_State)) +
  geom_bar(stat = "count", fill = "green") +
  theme_light() +
  labs(title = "Testing data Meter state distribution (nopca)")
i

j <- ggplot(data = train_pca, aes(x = Meter_State)) +
  geom_bar(stat = "count", fill = "green") +
  theme_light() +
  labs(title = "Training data Meter state distribution (pca)")
j

k <- ggplot(data = test_pca, aes(x = Meter_State)) +
  geom_bar(stat = "count", fill = "green") +
  theme_light() +
  labs(title = "Testing data Meter state distribution (pca)")
k






#train classifier

start_time_C5_nopca <- Sys.time()
set.seed(0)
C5_model_nopca <- train(Meter_State ~ .,
                    data = train_nopca,
                    method = "C5.0",
                    preProcess = c("center","scale"),
                    trControl = trctrl,
                    Metric = "Accuracy",
                    tuneGrid = expand.grid(.trials = c(1:20),
                                           .winnow = FALSE,
                                           .model = "tree"))

end_time_C5_nopca <- Sys.time()

#display model details
print(C5_model_nopca)

#predict values using C5.0 model
pred_C5.test <- predict(C5_model_nopca, test_nopca[,-ncol(test_nopca)], type = "raw")
pred_C5.train <- predict(C5_model_nopca, train_nopca[,-ncol(train_nopca)], type = "raw")

#evaluate classifier on training set
table(pred_C5.train, train_nopca$Meter_State)

cat("C5.0 model, without PCA, training set Accuracy; ",
    mean(pred_C5.train==train_nopca$Meter_State)*100,"%")

#evaluate classifier on test set
table(pred_C5.test, test_nopca$Meter_State)

cat("C5.0 model, without PCA, test set Accuracy; ", mean(pred_C5.test==test_nopca$Meter_State)*100,"%",




########################################C5.0 experiment with PCA##########################################



#train classifier
start_time_C5_pca <- Sys.time()
set.seed(0)
C5_PCA_model <- train(Meter_State ~ .,
                    data = train_pca,
                    method = "C5.0",
                    trControl = trctrl,
                    tuneGrid = expand.grid(.trials = c(1:20),
                                           .winnow = FALSE,
                                           .model = "tree"))
end_time_C5_pca <- Sys.time()

#display model details
print(C5_PCA_model)

#predict values using C5.0 model
pred_C5_PCA.test <- predict(C5_PCA_model, test_pca[,-ncol(test_pca)], type = "raw")
```

```r
pred_C5_PCA.train <- predict(C5_PCA_model, train_pca[,-ncol(train_pca)], type = "raw")

#evaluate classifier on training set
table(pred_C5_PCA.train, train_pca$Meter_State)

cat("C5.0 model, with PCA, training set Accuracy; ", mean(pred_C5_PCA.train==train_pca$Meter_State)*100

#evaluate classifier on test set
table(pred_C5_PCA.test, test_pca$Meter_State)

cat("C5.0 model, with PCA, test set Accuracy; ", mean(pred_C5_PCA.test==test_pca$Meter_State)*100,"%",



####################################SVM model without PCA#############################################

#train classifier

start_time_SVM_nopca <- Sys.time()
set.seed(0)
SVM_model_nopca <- train(Meter_State ~ .,
                    data = train_nopca,
                    method = "svmLinear",
                    tuneGrid = expand.grid( C = c(1:15)),
                    trControl = trctrl)
end_time_SVM_nopca <- Sys.time()

print(SVM_model_nopca)

#predict values using SVM model
pred_SVM.test <- predict(SVM_model_nopca, test_nopca[,-ncol(test_nopca)], type = "raw")
pred_SVM.train <- predict(SVM_model_nopca, train_nopca[,-ncol(train_nopca)], type = "raw")

#evaluate classifier on training set
table(pred_SVM.train, train_nopca$Meter_State)

cat("SVM model without PCA training set Accuracy; ", mean(pred_SVM.train==train_nopca$Meter_State)*100,


#evaluate classifier on test set
table(pred_SVM.test, test_nopca$Meter_State)

cat("SVM model without PCA test set Accuracy; ",
    mean(pred_SVM.test==test_nopca$Meter_State)*100,"%", "\n",
    "CPU time =", end_time_SVM_nopca - start_time_SVM_nopca, "seconds")



################################################SVM model with PCA########################################

#train classifier

start_time_SVM_pca <- Sys.time()
```

```r
set.seed(0)
SVM_PCA_model <- train(Meter_State ~ .,
                    data = train_pca,
                    method = "svmLinear",
                    tuneGrid = expand.grid( C = c(1:15)),
                    trControl = trctrl)
end_time_SVM_pca <- Sys.time()

print(SVM_PCA_model)

#predict values using SVM model
pred_SVM_PCA.test <- predict(SVM_PCA_model, test_pca[,-ncol(test_pca)], type = "raw")
pred_SVM_PCA.train <- predict(SVM_PCA_model, train_pca[,-ncol(train_pca)], type = "raw")

#evaluate classifier on training set
table(pred_SVM_PCA.train, train_pca$Meter_State)

cat("SVM model with PCA training set Accuracy; ", mean(pred_SVM_PCA.train==train_pca$Meter_State)*100,"


#evaluate classifier on test set
table(pred_SVM_PCA.test, test_pca$Meter_State)

cat("SVM model With PCA test set Accuracy; ",
    mean(pred_SVM_PCA.test==test_pca$Meter_State)*100,"%", "\n",
    "CPU time =", end_time_SVM_pca - start_time_SVM_pca, "seconds")



#######################################Comparison of results###########################################

# collect resamples
results <- resamples(list(C5.0_noPCA=C5_model_nopca,
                          C5.0_PCA=C5_PCA_model,
                          SVM_noPCA=SVM_model_nopca,
                          SVM_PCA=SVM_PCA_model))

#Plot resamples as a dotplot with 95% confidence interval
scales <- list(x=list(relation="free"), y=list(relation= "free"))
dotplot(results, scales=scales, conf.level = 0.95)

summary(C5_PCA_model$finalModel)
```