Supplementary Information
Autonomous analysis to identify bijels from two-dimensional images

Emily M. Gould, Katherine A. Macmillan, and Paul S. Clegg

Contents

1 K-nearest neighbours algorithm

The k-nearest neighbours (KNN) algorithm is one of the simplest and therefore most commonly used machine learning classification algorithms. This method classifies an unknown point in variable space based on a "vote" of its nearest k neighbours of known classification. A number of metrics can be used to determine the distance between points (and therefore which neighbours are the nearest), but the most common is the Euclidean distance [1]: the distance between points $\vec{p} = (p_1, p_2, ..., p_d)$ and $\vec{q} = (q_1, q_2, ..., q_d)$ in d dimensional space is given by

$$d(\vec{p}, \vec{q}) = \sqrt{\sum_{i=1}^{d} (q_i - p_i)^2}. \quad (1)$$

The number of neighbours k is tuned to improve the fit of the model. A higher value of k results in a smoother decision boundary between the different classes. A lower value of k will always give a lower training error; this can be understood by considering the extreme case of $k = 1$, where every training point is correctly classified by definition since its nearest neighbour is itself. However, this is very dependent on the particular training set and therefore not a useful measure of the performance of an algorithm. If instead we assess a validation error (obtained by using a test set or cross-validation) as a function of k, we expect a u-shaped curve. The best value of k can therefore be chosen as one that minimises the error. We often plot the accuracy, rather than error, as a function of k, in which case we look to maximise the accuracy.

In order to avoid inadvertently weighting some variables more than others, this algorithm requires that all variables are normalised so that they are of comparable scale. As long as this is done, the absolute scale of the variables is unimportant. The scale for normalisation is therefore often chosen as the mean of one arbitrary variable.

2 Logistic regression algorithm

In logistic regression, we classify points based on the probability $p(X)$ that a data point with input variables $X = (X_1, X_2, ...X_n)$ belongs to a given class. We achieve this by by fitting a linear equation (i.e. performing a linear regression) and then passing it through a logit function that ensures the output is bounded between 0 and 1, as a probability should be. The combined function,

$$\log\left(\frac{p(X)}{1-p(X)}\right) = \beta_0 + \beta_1 X_1 + ... + \beta_n X_n + \epsilon_b, \tag{2}$$

or

$$p(X) = \frac{e^{\beta_0+\beta_1 X_1+...+\beta_n X_n}}{1+e^{\beta_0+\beta_1 X_1+...+\beta_n X_n}} \tag{3}$$

is fitted to the training data using the maximum likelihood method (Equation 4) to find values of β that fit the training data provided [2], and therefore the probability. Here we assume a Bernoulli distribution for the residual error, which is more appropriate than a Gaussian for the case where the response is binary rather than continuous.

The maximum likelihood method is an alternative to the least squares method often used to fit linear regression models (in fact, when considering linear regression, the least squares approach is a special case of maximum likelihood [2]), and has better statistical properties than the least squares method. We find values for the coefficients β such that the predicted possibility for each data point corresponds as closely as possible to its known classification. This can be done by finding values that maximise the likelihood function:

$$l(\beta_0, \beta_1 ...\beta_n) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=0} (1 - p(x_{i'})) \tag{4}$$

where l is the likelihood function, x_i is the estimated probability that data point i belongs to the class, and y_i is the true probability of data point i belonging to the class. By definition, y_i must equal either 0 or 1 because the point is known to be either in the class or not. Therefore, the first product is a measure of how well the true positives are classified by the algorithm, and the second product a measure of how well the true negatives are classified.

3 Decision tree algorithm

The decision tree algorithm can be used for either classification or regression, and involves splitting the available variable space into different regions that share similar outcomes. To make predictions for a new data point, we look at which region it is in and predict its outcome to be the mean (for regression) or mode (for classification) of the training observations in that region. These methods are useful because they are simple and very easy to interpret.

In order to form a classification tree, we take an approach known as recursive binary splitting, wherein we start at the 'top' of the tree and split the predictor space in two, such that the classes are best separated. We then repeat the splitting process down both possible branches, and stop growing the tree when further branching leads to no significant improvement in the classification error. The final tree shows us the proportion of a particular category in each end node as well as the splits that have been chosen.

4 Efficient global optimisation of image pre-processing

This method allows us to find a global minimum in the classification error rate of a decision tree by efficiently exploring the parameter space defined by the possible pre-processing steps. Each combination of pre-processing steps, x, that may be attempted needs a predicted outcome, y, and a standard error on the

prediction, s. The current best set of pre-processing steps yields an outcome f_{min}. The expected improvement is given by

$$\mathrm{EI}(x) = (f_{min} - y)\Phi\left(\frac{f_{min} - y}{s}\right) + s\phi\left(\frac{f_{min} - y}{s}\right) \tag{5}$$

where ϕ and Φ are the normal probability density function and the normal cumulative distribution function, respectively [3, 4].

For a machine learning algorithm to carry out efficient global optimisation it must both make predictions of outcomes and calculate uncertainties on these predictions. A very natural choice is to use Gaussian process regression. A more common approach to supervised machine learning is to fit a single class of function to the available data, but this has obvious limitations if the functions chosen does not describe the data. Gaussian process regression, instead, gives a prior probability to very many functions that pass through the known points in the data. Roughly speaking the prior controls the smoothness of the functions and it can be learned from the training data. The Gaussian process approach can then be used to predict a value and an uncertainty for failure rate resulting from image pre-processing approaches that have not yet been attempted [5]. We explore the space of our four image pre-processing parameters in order to find the minimum value of the product of the error rate and the proportion of images successfully analysed. The latter is included to prevent the pre-processing step from minimising the error rate by making all images un-analysable.

5 Image pre-processing steps

We have optimized image pre-processing steps to improve our categorization of confocal micrographs from experiments. The sequence of pre-processing steps is illustrated in Figures S1 and S2 which show bijel and non-bijel samples respectively. The original liquid channel images are shown in (1) the left hand column. The subsequent columns show (2) the median filter (3) a band-pass filter (4) histogram equalization (5) grey scale threshold (6) shape filter (7) inversion (8) shape filter on the other domain. The parameters of (2), (3) and (6/8) are determined using efficient global optimization. This algorithm chose to discard the median filter entirely in its best performing sequence of steps.

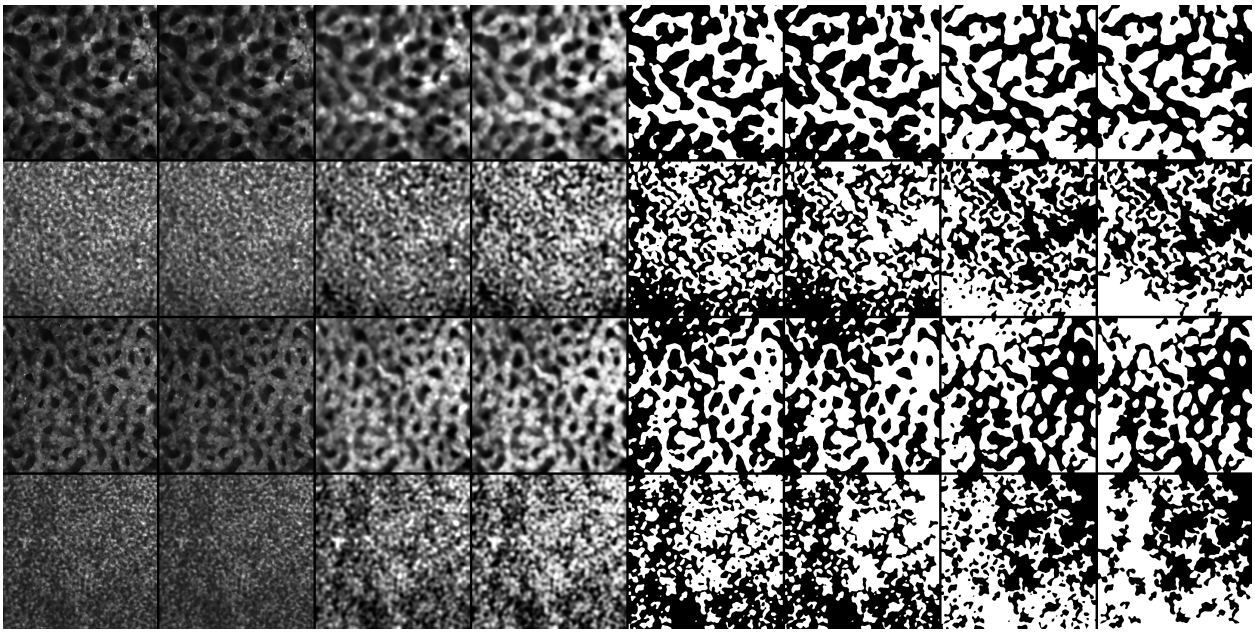


Figure S1: The effect of image pre-processing steps on several bijel images. The four samples are arranged vertically with the pre-processing steps arranged horizontally, as described in the text.

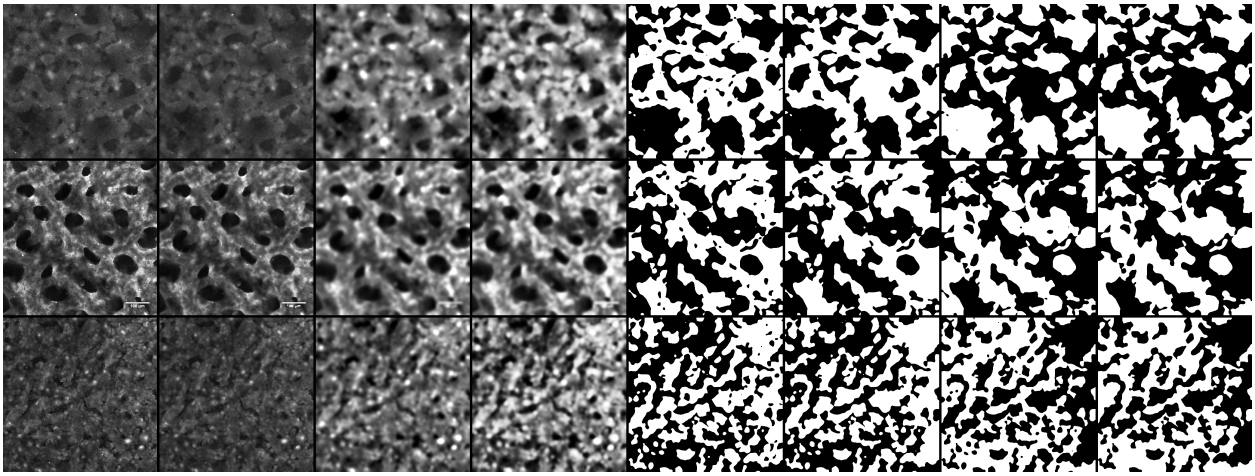


Figure S2: The effect of image pre-processing steps on several non-bijel images. The three samples are arranged vertically with the pre-processing steps arranged horizontally, as described in the text.

6 Walkthrough with examples in R

6.1 Getting the variables

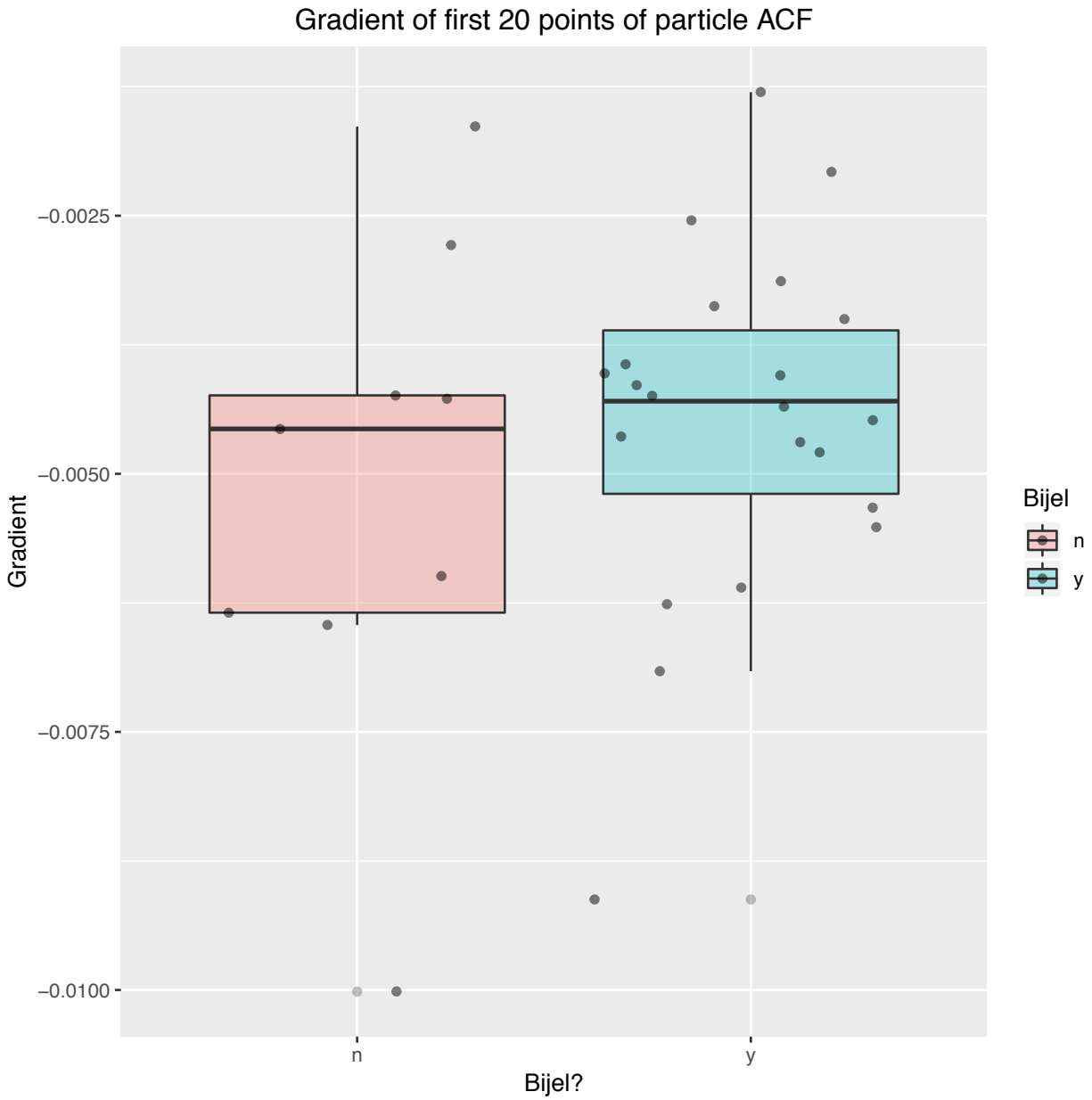
When performing this analysis on a set of data, we start with the autocorrelation functions of the two channels in each image, and labels of whether these images are bijels or not.

```
##      Sample.Number Bijel            ACF_1              ACF_2
## 19              19     n list(ACF_liquid) list(ACF_particle)
## 20i            20i     y list(ACF_liquid) list(ACF_particle)
## 20ii          20ii     y list(ACF_liquid) list(ACF_particle)
## 21              21     n list(ACF_liquid) list(ACF_particle)
## 22i            22i     n list(ACF_liquid) list(ACF_particle)
## 22ii          22ii     n list(ACF_liquid) list(ACF_particle)
```

We then need to turn these functions into a set of single-valued variables that describe features that may separate bijels from non-bijels, such as:

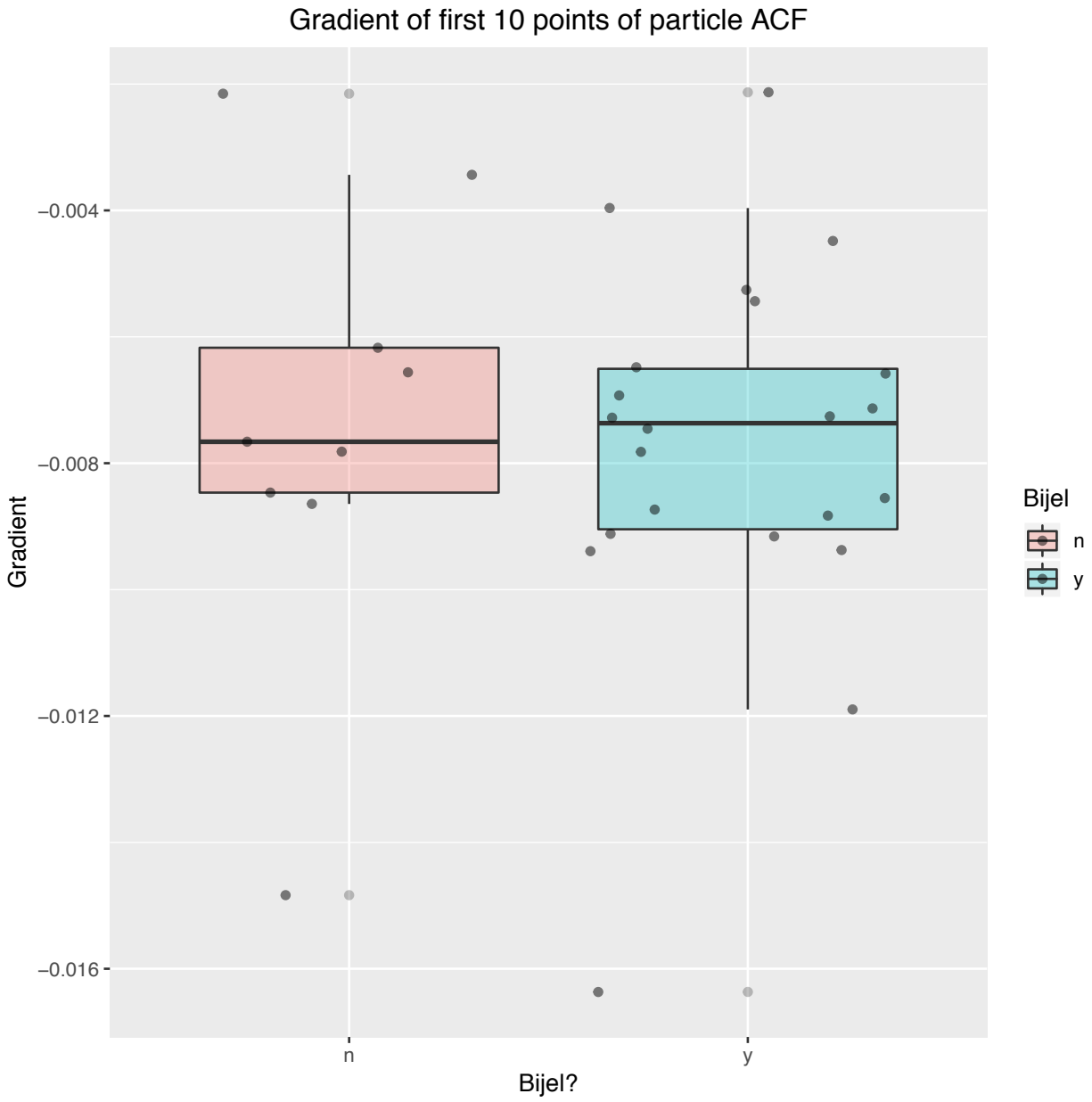
- The gradient of the particle channel autocorrelation function

```
r <- c(1:256)
num_points <- length(exp_Data$Sample.Number)
y <- exp_Data$Autocorrelation.Particle[1:20]
lineFits <- lapply(1:num_points,
                   function(n) lm(unlist(y[n,]) ~ r[1:20]))
lineCoeffs <- lapply(lineFits,
                     function(m) m$coefficients)
lineGradients <- lapply (1:num_points,
                         function(p) unname(lineCoeffs[[p]][2]))
exp_Data$Particle.Gradients.20 <- unlist(lineGradients)


library(ggplot2)
ggplot(exp_Data,
       aes(x=as.factor(Bijel), y=Particle.Gradients.20,
           fill=Bijel)) +
       geom_boxplot(alpha=0.3) +
       geom_jitter(alpha=0.5) +
       xlab("Bijel?") + ylab("Gradient") +
       ggtitle("Gradient of first 20 points of particle ACF") +
       theme(plot.title = element_text(hjust = 0.5))
```



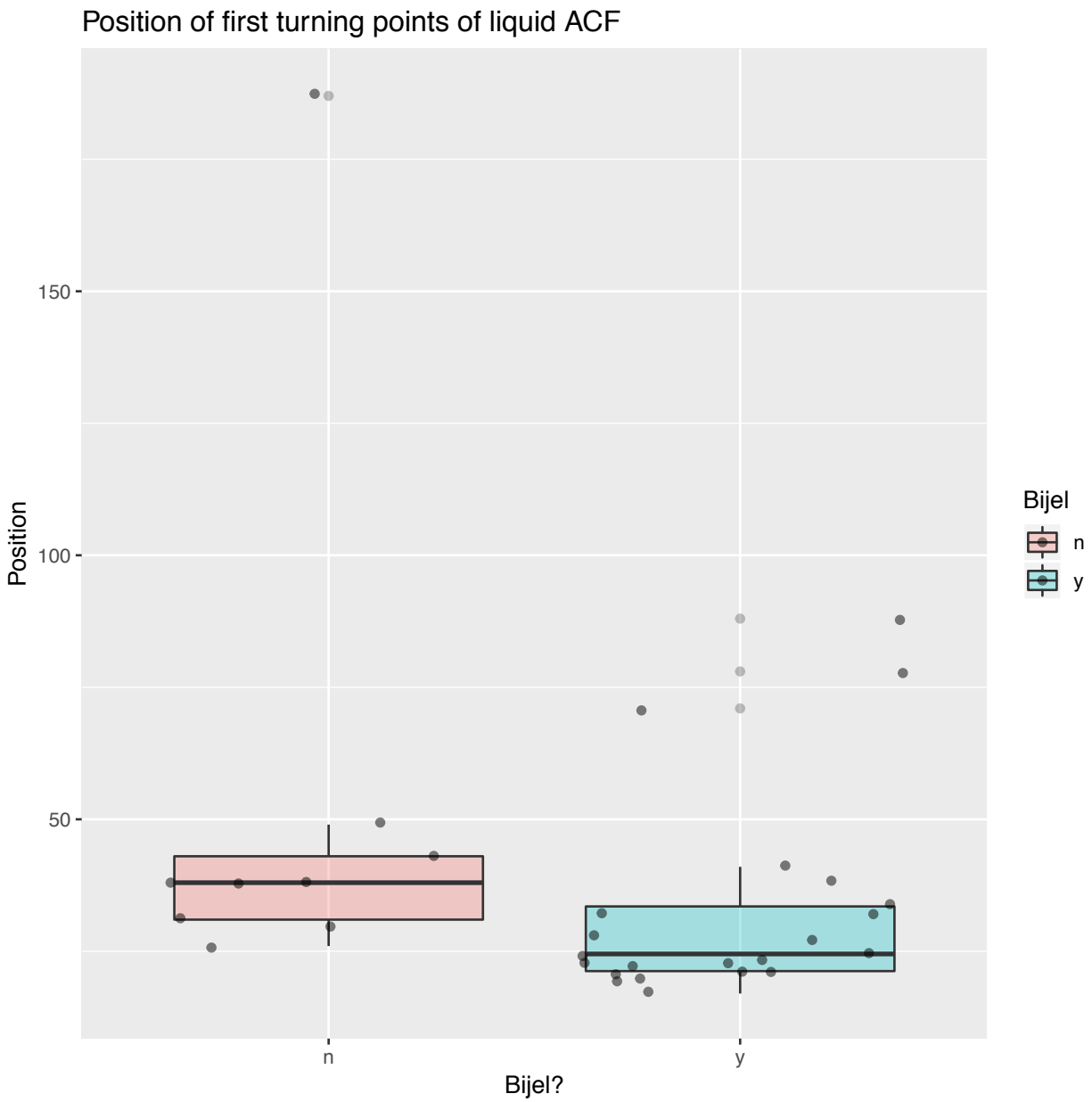

```
y2 <- exp_Data$Autocorrelation.Particle[1:10]
lineFits2 <- lapply(1:num_points,
                    function(n) lm(unlist(y2[n,]) ~ r[1:10]))
lineCoeffs2 <- lapply(lineFits2,
                      function(m) m$coefficients)
lineGradients2 <- lapply (1:num_points,
                          function(p) unname(lineCoeffs2[[p]][2]))
exp_Data$Particle.Gradients.10 <- unlist(lineGradients2)

ggplot(exp_Data,
       aes(x=as.factor(Bijel), y=Particle.Gradients.10,
```

```
        fill=Bijel)) +
    geom_boxplot(alpha=0.3) +
    geom_jitter(alpha=0.5) +
    xlab("Bijel?") + ylab("Gradient") +
    ggtitle("Gradient of first 10 points of particle ACF") +
    theme(plot.title = element_text(hjust = 0.5))
```



- The position of the first turning point in the liquid channel autocorrelation function

```
library(pastecs)
liquidTurns <- lapply(1:num_points,
                      function(y) turnpoints(unlist(
                        exp_Data$Autocorrelation.Liquid[y,])))
firstTurn <- lapply(1:num_points,
                    function(y) liquidTurns[[y]]$tppos[1])
exp_Data$Liquid.First.Turn <- unlist(firstTurn)

ggplot(exp_Data,
       aes(x=as.factor(Bijel), y=Liquid.First.Turn,
           fill=Bijel)) +
       geom_boxplot(alpha=0.3) +
       geom_jitter(alpha=0.5) +
       xlab("Bijel?") + ylab("Position") +
       ggtitle("Position of first turning points of liquid ACF")
```



We have generated box-and-jitter plots to see how the distribution of these variables is dependent on whether or not the sample is a bijel.

Once we have these variables, we can apply a machine learning model to it in one of two ways: training the model on the data and testing via cross-validation, and testing a previously trained model on the new data.

6.2 Training a model and testing with cross-validation

Here we have chosen to train a k-nearest neighbours model with the three variables shown above, based on our experience with a previous set of data. This can be done very simply using the CARET package in R, which contains a number of widely-used machine learning algorithms as well as cross-validation capabilities.

```r
library(caret)
```

```
## Loading required package:  lattice
```

```r
# set up the data
attach(exp_Data)
dat=data.frame(
  Particle.Gradients.20,
  Particle.Gradients.10,
  Liquid.First.Turn,
  Bijel)

# set a random number seed for reproducability
set.seed(1234)

# define the cross-validation parameters:
# here we use 10-fold cross-validation and repeat it 3 times
trCtrl <- trainControl(
  method = "repeatedcv",
  number = 10,
  repeats = 3)

knnFit <- train(Bijel~., # Bijel = output, other variables = input
                data=dat, # define the data
                method="knn", # choose the algorithm
                trControl=trCtrl, # cross-validation as above
                tuneLength=10)
print(knnFit)
```

```
## k-Nearest Neighbors
##
## 31 samples
##  3 predictor
##  2 classes: 'n', 'y'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 28, 28, 28, 28, 28, 27, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.6611111   0.13928571
##    7  0.5611111  -0.01034483
##    9  0.6638889   0.19310345
##   11  0.7055556   0.22758621
##   13  0.7388889   0.25000000
##   15  0.7000000   0.00000000
##   17  0.7166667   0.00000000
##   19  0.7166667   0.00000000
##   21  0.7166667   0.00000000
##   23  0.7166667   0.00000000
##
```

```
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 13.
```

6.3 Testing a trained model on new data

If we have already trained a model, we can use it to classify new data. This is how the algorithm can be used as a tool to classify unlabelled data.

```
# predict whether the data points are bijels or not
# to do this, we have to omit the bijel label from the data
bijel_pred = predict(trainedKNN, newdata = dat[,-4])
bijel_true = dat[,4]
print(data.frame(bijel_pred, bijel_true=bijel_true))
```

```
##    bijel_pred bijel_true
## 1           y          n
## 2           y          y
## 3           y          y
## 4           y          n
## 5           y          n
## 6           y          n
## 7           y          y
## 8           y          y
## 9           y          n
## 10          y          n
## 11          y          y
## 12          y          y
## 13          y          n
## 14          n          n
## 15          n          n
## 16          n          y
## 17          y          y
## 18          y          y
## 19          y          y
## 20          y          y
## 21          y          y
## 22          y          y
## 23          y          y
## 24          y          y
## 25          n          y
## 26          n          y
## 27          y          y
## 28          n          y
## 29          y          y
## 30          n          y
## 31          y          y
```

We can now calculate the error rate, and also look at the results to see how many false positives, false negatives, etc. we have.

```
success_count=length(bijel_pred[bijel_pred==bijel_true])
success_rate=success_count/length(bijel_pred)
paste0("Success rate: ",round(100*success_rate),"%")

## [1] "Success rate: 61%"

library(gmodels)
CrossTable(x=bijel_pred, y=bijel_true, prop.chisq=FALSE)

##
##
##    Cell Contents
## |-------------------------|
## |                       N |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
##
##
## Total Observations in Table:  31
##
##
##              | bijel_true
##   bijel_pred |         n |         y | Row Total |
## -------------|-----------|-----------|-----------|
##            n |         2 |         5 |         7 |
##              |     0.286 |     0.714 |     0.226 |
##              |     0.222 |     0.227 |           |
##              |     0.065 |     0.161 |           |
## -------------|-----------|-----------|-----------|
##            y |         7 |        17 |        24 |
##              |     0.292 |     0.708 |     0.774 |
##              |     0.778 |     0.773 |           |
##              |     0.226 |     0.548 |           |
## -------------|-----------|-----------|-----------|
## Column Total |         9 |        22 |        31 |
##              |     0.290 |     0.710 |           |
## -------------|-----------|-----------|-----------|
##
##
```

In this case we can see that the error rate obtained from applying the old model to new data (39%) is much higher than the one obtained by directly training the same type of model on the data in question (13%). This is because the two datasets are from slightly different physical systems so although the same variables are useful, bijels are indicate at different values of these variables.

References

[1] K. P. Murphy, *Machine Learning : A Probabilistic Perspective*. Cambridge, MA: MIT press, 2012.

[2] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An Introduction to Statistical Learning*, vol. 103 of *Springer Texts in Statistics*. New York, NY: Springer New York, 2013.

[3] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient Global Optimization of Expensive Black-Box Functions," *Journal of Global Optimization*, vol. 13, no. 4, pp. 455–492, 1998.

[4] P. I. Frazier, "A Tutorial on Bayesian Optimization," jul 2018.

[5] C. E. Rasmussen and C. K. I. Williams, *Gaussian processes for machine learning*. Cambridge, MA: MIT press, 2006.