Lab 06 - Explain ML Models using LIME

Learning Outcomes

In this lab you will learn

- how to use the LIME package to
  - explain model predictions;
  - help us find the (possibly) best single variable model;
- how the logistic regression classifier works.

This labsheet is a long labsheet spanned over multiple weeks. It is a practice tutorial covering LIME which we don't have room to fit into our lectures. You should create an R file along the way to copy and run the code given below. There are small exercises that require you to write some R code also.

This labsheet is very relevant to what you need to do for project. So do go through it and try the exercises.

Local interpretable model-agnostic explanations (LIME)

Motivation

The improved prediction performance of modern machine learning methods like deep learning, random forests, SVMs, and gradient boosted trees comes at the cost of decreased explanation. On the other hand, a human domain expert can review the "if-then" structure of a decision tree and compare it to their own decision-making processes. From the comparison, the human expert can asses whether the decision tree will make reasonable decisions.

"Why should I Trust you" is one of the most challenging questions a data scientist needs to answer when deploying a model for real-world applications. How do we know our machine learning models are learning the concepts, and not just data quirks? There are quite a few examples on how machine learning are learning the "quirks" rather than the true concepts:

"A recent experiment at the University of Washington sought to create a classifier capable of distinguishing between images of Wolves from Huskies. The system was trained on a number of images and tested on an independent set of images as it is usually the case in machine learning. Surprisingly, even though Huskies are very similar to Wolves, the system managed to obtain around 90% accuracy. The researchers were ecstatic with such a result. However, on running an explainer function capable of explain why the algorithm managed to obtain such good results, it became immediately evident that the model was basing its decisions primarily on the background. Wolf

images usually had a snowy background, while husky images rarely did. So rather than creating a Wolves Vs Huskies classifier, the researchers had unwittingly created an amazing snow detector.” – Blog Article

Logistic Regression Models

Let’s use the *iris* dataset as an example.

Step 1: Loading the iris dataset

Load the dataset and divide it into a training set and a test set. The *iris* dataset contains 3 species. We will use a logistic regression classifier for binary classification: “setosa” (value 1) vs “non-setosa” (value 0).

```
iris <- iris
iris$class <- as.numeric(iris$Species == "setosa")
set.seed(12345)
intrain <- runif(nrow(iris)) < 0.75
train <- iris[intrain,]
test <- iris[!intrain,]
head(train)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species class
## 1          5.1         3.5          1.4         0.2  setosa     1
## 5          5.0         3.6          1.4         0.2  setosa     1
## 6          5.4         3.9          1.7         0.4  setosa     1
## 7          4.6         3.4          1.4         0.3  setosa     1
## 8          5.0         3.4          1.5         0.2  setosa     1
## 9          4.4         2.9          1.4         0.2  setosa     1
cat("Training set size is", dim(train))
## Training set size is 107 6
cat("Test set size is", dim(test))
## Test set size is 43 6
```

Step 2: Fitting a logistic regression model

```
responses <- colnames(iris) %in% c("Species", "class")
features <- colnames(iris)[!responses]

formula <- paste("class", paste(features, collapse=" + "), sep=" ~ ")
```

Inspect the variable `formula` you should see that we have just defined the response variable `class` to be dependent on the other variables:

```
formula
```

Next, we can call the `glm()` (*generalised linear model*) function to fit a linear model to our training set and use the trained model to make predictions on the training and test sets.

```
model_logr <- glm(formula=formula, data=train, family=binomial(link="logit"))
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
train$pred <- predict(model_logr, newdata=train, type="response")
test$pred <- predict(model_logr, newdata=test, type="response")
```

The warning messages from `glm.fit` above can be ignored. See the bottom of the sheet for another example and explanation.

Step 3: Performance evaluation

Define a function which takes in the predictions (`pred`), the ground truth (`truth`), and model name as input arguments. The function then computes the normalised *deviance*, *accuracy*, *precision*, *recall*, and *F1 score* and puts them into a data frame.

```
performanceMeasures <- function(pred, truth, name = "model") {
   ctable <- table(truth = truth, pred = (pred > 0.5))
   accuracy <- sum(diag(ctable)) / sum(ctable)
   precision <- ctable[2, 2] / sum(ctable[, 2])
   recall <- ctable[2, 2] / sum(ctable[2, ])
   f1 <- 2 * precision * recall / (precision + recall)
   data.frame(model = name, precision = precision,
              recall = recall,
              f1 = f1, accuracy = accuracy)
}
```

We can then write another function to print out and compare the model's performance on the training and test datasets.

```
pretty_perf_table <- function(model,training,test) {
  library(pander)
  # setting up Pander Options
  panderOptions("plain.ascii", TRUE)
  panderOptions("keep.trailing.zeros", TRUE)
  panderOptions("table.style", "simple")
  perf_justify <- "lrrrr"
  # comparing performance on training vs. test
  pred_train <- predict(model, newdata=training)
  truth_train <- training[, "class"]
  pred_test <- predict(model, newdata=test)
  truth_test <- test[, "class"]
  trainperf_tree <- performanceMeasures(
      pred_train, truth_train, "logistic, training")
  testperf_tree <- performanceMeasures(
      pred_test, truth_test, "logistic, test")
  perftable <- rbind(trainperf_tree, testperf_tree)
  pandoc.table(perftable, justify = perf_justify)
}
```

Now let's observe the performance.

```
pretty_perf_table(model_logr, train, test)
##
##
## model                   precision   recall   f1   accuracy
## -------------------- ----------- -------- ---- ----------
## logistic, training              1        1    1          1
## logistic, test                  1        1    1          1
```

Our logistic regression classifier achieves a perfect score for all the evaluation measures! We can also examine a few cases in our test set:

```
library(knitr)

cases <- c(5,11,13,24,30)
test[cases, c("class","pred")]
##     class         predR
## 20      1 1.000000e+00
## 43      1 1.000000e+00
## 51      0 2.220446e-16
## 89      0 2.220446e-16
## 113     0 2.220446e-16
```

Exercises:

1. Write a function called `confusion_matrix` which should take in two vectors `ytrue` and `ypred` as input arguments and return the confusion matrix as a data frame. It is quite easy to get the confusion matrix using the `table()` function (see Week 8's lecture note).
2. Repeat the above training and evaluation process for the **SPAM** dataset.

After you have written the `confusion_matrix()` function, you should be able to get the following confusion matrix for the logistic regression classifier on the test set:

```
confusion_matrix(test$class, test$pred > 0.5)
##      prediction
## truth FALSE TRUE
##     0    31    0
##     1     0   12
```

LIME – Automated Sanity Checking

Domain experts manually sanity-check a model by manually going through some example cases and inspecting the answers. Generally, you would want to try a few typical cases, and a few extreme cases, just to see what happens. You can think of LIME as one form of automated sanity checking.

Use LIME

Using the classic *iris* dataset above as an example again and our objective is to predict whether a given iris is a setosa species or not, based on the four features: petal and sepal length and width.

Suppose that the performance is extremely good with perfect predictions, but which feature plays a key role in determining whether a particular iris instance is setosa or not? We can see from the `summary()` function of a logistic regression model, we can get an overall answer. Regression models, together with basic decision trees are considered as interpretable models for domain experts to make sense of. More complex models then require special techniques for explanation. Out of the box, LIME supports the following model objects:

- `train` from the library *caret*
- `WrappedModel` from *mlr*
- `xgb.Booster` from *xgboost*
- `H2OModel` from *h2o*
- `keras.engine.training.Model` from *keras*
- `lda` from *MASS* (used for low-dependency examples)

XGBoost (Optional)

We won't be formally introduce the `XGBoost` classifier in this unit, the code below is just a glimpse on how to use this high performing ensemble tree-based method. For more information, please refer to the "Practical data science with R" book, second edition, Chapter 10. Below is the code extracted from Section 6.3.2 of the book.

```
# if needed, install the library xgboost
# install.packages("xgboost")
library(xgboost)

# Input:
# - variable_matrix: matrix of input data
# - labelvec: numeric vector of class labels (1 is positive class)
#
# Returns:
# - xgboost modelR
fit_iris_example = function(variable_matrix, labelvec) {
  cv <- xgb.cv(variable_matrix, label = labelvec,
               params=list(
                 objective="binary:logistic"
               ),
               nfold=5,
               nrounds=100,
               print_every_n=10,
               metrics="logloss")

  evalframe <- as.data.frame(cv$evaluation_log)
  NROUNDS <- which.min(evalframe$test_logloss_mean)

  model <- xgboost(data=variable_matrix, label=labelvec,
                   params=list(
                     objective="binary:logistic"
                   ),
                   nrounds=NROUNDS,
                   verbose=FALSE)

  model
}

# Fit the iris dataset using xgboost:
input <- as.matrix(train[features])
model <- fit_iris_example(input, train$class)
## [1]  train-logloss:0.455851+0.000089 test-logloss:0.455942+0.000886
## [11] train-logloss:0.033041+0.000072 test-logloss:0.033115+0.000717
## [21] train-logloss:0.023723+0.000067 test-logloss:0.023887+0.001041
## [31] train-logloss:0.023715+0.000065 test-logloss:0.023889+0.001136
## [41] train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001147
## [51] train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001148
## [61] train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001148
## [71] train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001148
## [81] train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001148
## [91] train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001148
## [100]    train-logloss:0.023715+0.000065 test-logloss:0.023890+0.001148
```

The following code uses the `lime()` function to build an **explainer**, which takes the feature columns of the training dataset, the model, and puts the continuous variables into 10 bins (`n_bins = 10`) when making explanations.

```
library(lime)
explainer <- lime(train[features], model = model,
                  bin_continuous = TRUE, n_bins = 10)
```

Now we can explain the model's prediction on test examples. Note that the `dplyr` package also has a function called `explain()`, so if you have `dplyr` in your namespace also, then you will get a conflict trying to call lime's `explain()` function. To prevent this ambiguity, you can specify the function using namespace notation: `lime::explain(...)`. The `explain()` function takes in the instances (i.e., the observations), the explainer built earlier by the `lime()` function, the number of labels to explain (use value `1` for binary classification), and the number of features (in this case `4`) to use when fitting the explanation.

```
cases <- c(3,11,21,30)
(example <- test[cases,features])
##     Sepal.Length Sepal.Width Petal.Length Petal.Width
## 4            4.6         3.1          1.5         0.2
## 43           4.4         3.2          1.3         0.2
## 77           6.8         2.8          4.8         1.4
## 113          6.8         3.0          5.5         2.1
explanation <- lime::explain(example, explainer, n_labels = 1, n_features = 4)
plot_features(explanation)
```

We can see from the plot that `Petal.Length` is the determining feature for the classification of these four iris instances. The rest of the features are less significant (where the blues are supporting features, reds are against).

For more information on how LIME works (which creates synthetic data points and work out a linear model closest to the data point to be explained), please see the excerpt of the (Second Edition) textbook (page 201).

Exercises:

1. In the code above, we used `lime()` to create an `explainer` variable using a *gradient boosting classifier* model (variable `model`) that was trained on our *iris* training set. We could replace the argument `model` by `model_logr` and then call the `lime::explain()` function to generate the explanation for the same four test cases (see variable `cases` above). Unfortunately, `lime()` accepts a limited number of model types. To overcome this issue, train your `model_logr` using the `caret::train` function describerd in Section 6 below. You should then be able to use `lime()` to generate an *explainer* object. You should see that the **Explanation Fit** values for each test case from the two models (`model` and `model_logr`) are not exactly the same, but `Petal.Length` should remain at the top, having the highest explanation fit value.
2. Copy the functions for computing the AUC (*area under curve*) values and for plotting the ROC curves from weeks 9 and 10's lecture note to your R code file for this labsheet.
3. Repeat the above process using the `Petal.Length` feature only. How does this new logistic regression classifier model perform for this single feature? How does it perform if you use the `Sepal.Width` feature instead? Compare the performances of these two new models with the one that uses all the four features. Show their performances using confusion matrices and on the same ROC plot.

Reference

Practical Data Science with R, *Nina Zumel, John Mount*, Manning, 2nd Ed., 2020, Section 6.3.

A closer look at `model_logr`

Let's look at the output from `summary(model_logr)`:

```
summary(model_logr)
##
## Call:
## glm(formula = formula, family = binomial(link = "logit"), data = train)
##
## Deviance Residuals:
##        Min          1Q      Median          3Q         Max
## -2.714e-05  -2.110e-08  -2.110e-08   2.110e-08   2.619e-05
##
## Coefficients:
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -10.103 514873.486       0        1
## Sepal.Length     10.246 154674.123       0        1
## Sepal.Width       7.615  73850.693       0        1
## Petal.Length    -18.911 128994.033       0        1
## Petal.Width     -23.797 193141.306       0        1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1.3922e+02  on 106  degrees of freedom
## Residual deviance: 2.3057e-09  on 102  degrees of freedom
## AIC: 10
##
## Number of Fisher Scoring iterations: 25
```

What are the coefficients?

Logistic regression is a **linear regression** technique which involves fitting a line (to be more specific, if there are two feature variables (2D) then it is indeed a **line**; if there are three feature variables (3D) then it is a **plane**; for higher dimensions (as in this case, 4D), it is a **hyperplane**) to the data and then converting the response values into probabilities using the `logistic` function, which is defined as:

$$\text{logistic}(x) = \frac{1}{1+e^{-(\beta_0+\beta_1 x)}}$$

From the output of the `summary()` function, we can see that the coefficients of the line that is fitted to separate the two classes can be extracted from the `coefficients` attribute. Let c_0 be the intercept and c_1 to c_4 be the coefficients for the four features `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width`. The equation of the line that separates the two classes is simply $c_0 + c_1 \times \text{Sepal.Length} + c_2 \times \text{Sepal.Width} + c_3 \times \text{Petal.Length} + c_4 \times \text{Petal.Width} = 0.$ When substituting the four feature values of instances into the equation, instances belonging to the same class should fall onto the same side of this line but instances belonging to different classes should be on different sides of this line, as shown in the code below:

```
coef <- model_logr$coefficients
# separate the coefficients for the 4 feature variables from the intercept
coef0 <- coef[1]   # intercept
coef <- coef[-1]

# compare the values stored in the response variable below and the ground truth
# label (test[,"class"]). Recall that %*% denotes matrix multiplication.
response <- as.matrix(test[,features]) %*% coef + coef0  # this is a column vector
print(as.vector(response))             # convert into a row vector for printing
##  [1]  31.71056  33.07547  27.50730  32.96067  35.58074  40.77002  27.66535
##  [8]  43.55637  38.65912  36.05389  30.00179  23.24161 -36.21228 -40.95719
## [15] -33.62961 -31.31641 -47.71775 -29.65911 -50.67662 -33.61883 -43.19838
## [22] -33.51380 -42.42713 -38.35297 -45.71694 -45.06520 -76.06134 -84.17602
## [29] -75.19889 -71.57075 -82.91410 -75.12328 -67.50534 -68.26581 -58.86450
## [36] -61.24725 -81.46270 -67.02179 -78.45551 -67.89364 -71.68149 -66.28621
## [43] -77.54051
print(test$class)
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
## [39] 0 0 0 0 0
# we can see that the responses are of opposite signs for instances that belong to differ
```

We can use the `sigmoid()` function from the `e1071` library to convert the response values above to probabilities. The `sigmoid()` function is the same as the logistic function except that β_0 and β_1 are set to 0 and 1 respectively. Using the `sigmoid()` function, our manually computed probabilities are:

```
# the sigmoid function is defined in this library
library(e1071)

# convert responses into probabilities using sigmoid()
pred_manual <- as.vector(sigmoid(response))
cat("Predictions (as probability values) computed manually:")
## Predictions (as probability values) computed manually:
print(pred_manual)
##  [1] 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00
##  [6] 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00
## [11] 1.000000e+00 1.000000e+00 1.875880e-16 1.631234e-18 2.482247e-15
## [16] 2.508753e-14 1.889932e-21 1.315873e-13 9.804449e-23 2.509148e-15
## [21] 1.734531e-19 2.787023e-15 3.750863e-19 2.205559e-17 1.397601e-20
## [26] 2.681833e-20 9.267874e-34 2.772173e-37 2.195518e-33 8.264564e-32
## [31] 9.791844e-37 2.367965e-33 4.817277e-30 2.251820e-30 2.725654e-26
## [36] 2.515694e-27 4.180230e-36 7.812826e-30 8.456807e-35 3.267126e-30
## [41] 7.398185e-32 1.630297e-29 2.111474e-34
cat("Predictions (as probability values) from the logistic regression classifier implemen
## Predictions (as probability values) from the logistic regression classifier implemente
print(test$pred)
##  [1] 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00
##  [6] 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00 1.000000e+00
## [11] 1.000000e+00 1.000000e+00 2.220446e-16 2.220446e-16 2.220446e-16
## [16] 2.220446e-16 2.220446e-16 1.315873e-13 2.220446e-16 2.220446e-16
## [21] 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16
## [26] 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16
## [31] 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16
## [36] 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16 2.220446e-16
## [41] 2.220446e-16 2.220446e-16 2.220446e-16
```

The two vectors `pred_manual` and `test$pred` are not exactly the same because we we didn't use the optimal β_0 and β_1 values estimated by `glm()` in the model fitting process; however, these two vectors look very similar and their differences are well beyond the 10th decimal place and can be considered to be negligible.

The verification above helps us understand how the logistic regression classifier algorithm works under the hood.

How are the deviances, degrees of freedom, and AIC computed?

We also notice from the summary the *null deviance*, *residual deviance*, *degrees of freedom*, and *AIC* values:

```
model_logr$null.deviance
## [1] 139.2221
model_logr$df.null
## [1] 106
model_logr$deviance
## [1] 2.305677e-09
model_logr$df.residual
## [1] 102
model_logr$aic
## [1] 10
```

How are they computed?

In our case, the null model simply returns a fixed probability value that is equal to the proportion of setosa (`train$class == 1`) in the training set. That is,

```
prob.null <- sum(train$class == 1) / nrow(train)
print(prob.null)
## [1] 0.3551402
```

From the lecture note, the deviance is defined as $-2 \times (\text{logLikelihood} - S)$, where S is called the log-likelihood of the *saturated model* and is equal to zero in most cases. The log likelihood of our model is given by $\sum_{i=1}^{N} y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$ where y_i denotes the ground truth class label (1 for setosa and 0 for non-setosa) for the i^{th} instance; p_i denotes the output probability estimated by the algorithm for the i^{th} instance; and N is the size of the dataset. Although there are two terms inside the summation, only one term is active at a time: when $y_i = 1$, the second term is 0 ; when $y_i = 0$, the first term is 0 .

A good algorithm should output a small probability value (close to 0) when $y_i = 0$ and a large probability value (close to 1) when $y_i = 1$. For our null model here, p_i is equal to the value of `prob.null` above for all i .

```
# make sure that we ignore the NAs when we do the summation.
deviance.null <- -2 * sum(train$class*log(prob.null) + (1-train$class)*log(1-prob.null),
cat('Our computed null deviance is:', deviance.null)
## Our computed null deviance is: 139.2221
```

Our computed null deviance value agrees with the null deviance displayed by the `summary()` function.

For our logistic regression classifier model `model_logr`, we have a different probability p_i for each training instance:

```
# compute the responses of instances for the training set
response <- as.vector( as.matrix(train[,1:4]) %*% coef + coef0 )
# convert to probabilities using sigmoid()
prob <- sigmoid(response)
deviance.logr <- -2 * sum(train$class*log(prob) + (1-train$class)*log(1-prob), na.rm=TRUE
cat('Our manually computed deviance for the logistic regression classifier model is:', de
## Our manually computed deviance for the logistic regression classifier model is: 2.3060

# alternatively, we can use the probabilities predicted by model_logr in Section 3 above
deviance.logr <- -2 * sum(train$class*log(train$pred) + (1-train$class)*log(1-train$pred)
cat('Our manually computed deviance for the logistic regression classifier model is:', de
## Our manually computed deviance for the logistic regression classifier model is: 2.3056
```

We can see that the first estimate is very close to the `model_logr$deviance` value and the second estimate is identical to it.

The null model collates all the data to get the single parameter `prob.null` for the classification task, it therefore has degrees of freedom equal to the training set size minus 1. The logistic regression classifier model `model_logr` uses **5** coefficients (`coef0`, `coef1`, ..., `coef4`) as parameters for classification, its degrees of freedom thus equal to the training set size minus 5. From week 10's lecture note, $\mathrm{AIC} = \mathrm{deviance} + 2 \times \mathrm{numberOfParameters}$. This can be easily computed.

```
cat('Degrees of freedom of the null model:', nrow(train)-1)
## Degrees of freedom of the null model: 106
df.logr <- nrow(train)-5            # number of parameters for model_logr is 5
cat('Degrees of freedom of the model_logr:',df.logr)
## Degrees of freedom of the model_logr: 102
aic.logr <- deviance.logr + 2*5     # number of parameters for model_logr is 5
cat('AIC for model_logr is:', aic.logr)
## AIC for model_logr is: 10
```

Now we know how the deviance values, degrees of freedom, and AIC are computed in `model_logr`. We should note that all these values are computed for the training set, as we don't alter the model after it is trained. Of course, knowing the formulae for computing these values means that we can compute them for the calibraton and test sets as well. However, we should keep in mind that these values depend on the size of the dataset. As the calibration (or test) set is much smaller, we should see a drop in the deviance values for both `model_logr` and the null model but it doesn't mean they perform better on the calibration (or test) set. Instead of the *deviance*, the **normalised deviance** is therefore often used (See Listing 10.1, page 357 of the textbook in the **Reference** section above).

Exercise:

- Compare the deviances, degrees of freedom, and AIC values of `model_logr` and the two single variable classifiers using `Petal.Length` and `Sepal.Width`.

The `caret` library

The `caret` library supports a large number of machine learning algorithms for classification and regression tasks. To train for a model, use the `train()` function; to predict, use the `predict()` function. For example,

```
# using two features
features <- c("Petal.Length", "Petal.Width")
# training - like the glm() function above, you should see warning messages.
# since logistic regression classifier is a linear method, we can use the
# glm method.
model <- caret::train(x = train[features], y = train[,"class"],
               method = "glm", family = binomial(link="logit"),
               metric = "Accuracy")
# making prediction
pred <- predict(model, test[features])
# using LIME
explainer <- lime(train[features], model=as_classifier(model),
    bin_continuous=TRUE, n_bins=10)
```

Note: `caret::train()` expects the argument for `x` to be a data frame. So if you use a single feature, say `Petal.Length`, to train the model, then ensure you code the function call as `caret::train(x = train["Petal.Length"], ...)` or `caret::train(x = train[,"Petal.Length", drop=FALSE], ...)` so that the single column would not collapse to a vector.

To find out what machine learning methods are supported by this library, type:

```
models_table <- modelLookup()
```

This is a large table having more than 500 rows. It has 6 columns. The names of the supported methods are in column 1.

Advanced Exercise:

- Try one of the following machine learning methods: support vector machine or decision trees. Both are supported by the `caret` library.

Working on Project

Follow the Week 9's lecture notes to build single variable models for both categorical and numerical variables.

Warning messages from `glm.fit`

Let's consider the simple example below:

```
x <- rnorm(100)
y <- c(rnorm(50, mean=5), rnorm(50, mean=-5))
res <- c(rep(0, 50), rep(1, 50))
df <- data.frame(x=x, y=y, response=as.factor(res))

library(ggplot2)
ggplot(df) + geom_point(aes(x=x, y=y, colour=response, shape=response), size=3)
```

We can see that the two clusters of points are far apart from each other and can be easily separated by a line. This is an easy binary classification problem, with the first class having value 0 and the second class having value 1 under the `response` column of the data frame `df`. Let's try using the logistic regression classifier to classify the two classes.

```
model_simple <- glm(formula = response ~ x + y, family = binomial(link='logit'), data = d
## Warning: glm.fit: algorithm did not converge
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

and we see the same warning messages. It seems that `glm.fit` tried to iteratively optimise for the solution (the linear decision boundary that separates the two classes) but when it got a perfect solution in the first iteration (so no need to iterate further), it produced warning messages. Same for the *iris* dataset above, the classes are easily separable. It is safe to ignore the warning messages there also.

Previous
Lab 05 - Reading Files and Web Scrapping using R

Next
Lab 07 - Single Variable Models and Naïve Bayes model

Last updated 2 hours ago