SC1_Proj

Alessio

13 January 2020

Introduction

One of the most common types of cancer diagnosed in women is breast cancer. There are multiple tests that people are subjected to, but one of the most indicative ones is fine needle aspiration which involes extracting a sample of cells to be examined under a microscope. Multiple numerical metrics are computed from the obtained images. The aim is to use the extracted metrics to make accurate diagnoses.

The dataset consists of 569 images which have been processed as described and a total of 30 variables have been computed for each observation.

The aim of this report is to implement a number of classification algorithms, use them to obtain predictions, and compare their performances.

Exploratory analysis

```
data <- read_csv("../data/data.csv")
colnames(data)[3:32] <- c(
  "radius_m", "texture_m", "perim_m", "area_m", "smooth_m", "compact_m",
  "concav_m", "concav_pt_m", "symmetry_m", "frac_dim_m", "radius_se",
  "texture_se", "perim_se", "area_se", "smooth_se", "compact_se",
  "concav_se", "concav_pt_se", "symmetry_se", "frac_dim_se", "radius_w",
  "texture_w", "perim_w", "area_w", "smooth_w", "compact_w", "concav_w",
  "concav_pt_w", "symmetry_w", "frac_dim_w"
)
```

Check for missing values in every column.

```
colSums(is.na(data))
```

```
##           id    diagnosis     radius_m    texture_m      perim_m       area_m
##            0            0            0            0            0            0
##     smooth_m    compact_m     concav_m  concav_pt_m   symmetry_m   frac_dim_m
##            0            0            0            0            0            0
##    radius_se   texture_se     perim_se      area_se    smooth_se   compact_se
##            0            0            0            0            0            0
##    concav_se concav_pt_se  symmetry_se  frac_dim_se     radius_w    texture_w
##            0            0            0            0            0            0
##      perim_w       area_w     smooth_w    compact_w     concav_w  concav_pt_w
##            0            0            0            0            0            0
##   symmetry_w   frac_dim_w          X33
##            0            0          569
```

```
data %<>% mutate_at(vars(diagnosis), factor)
```

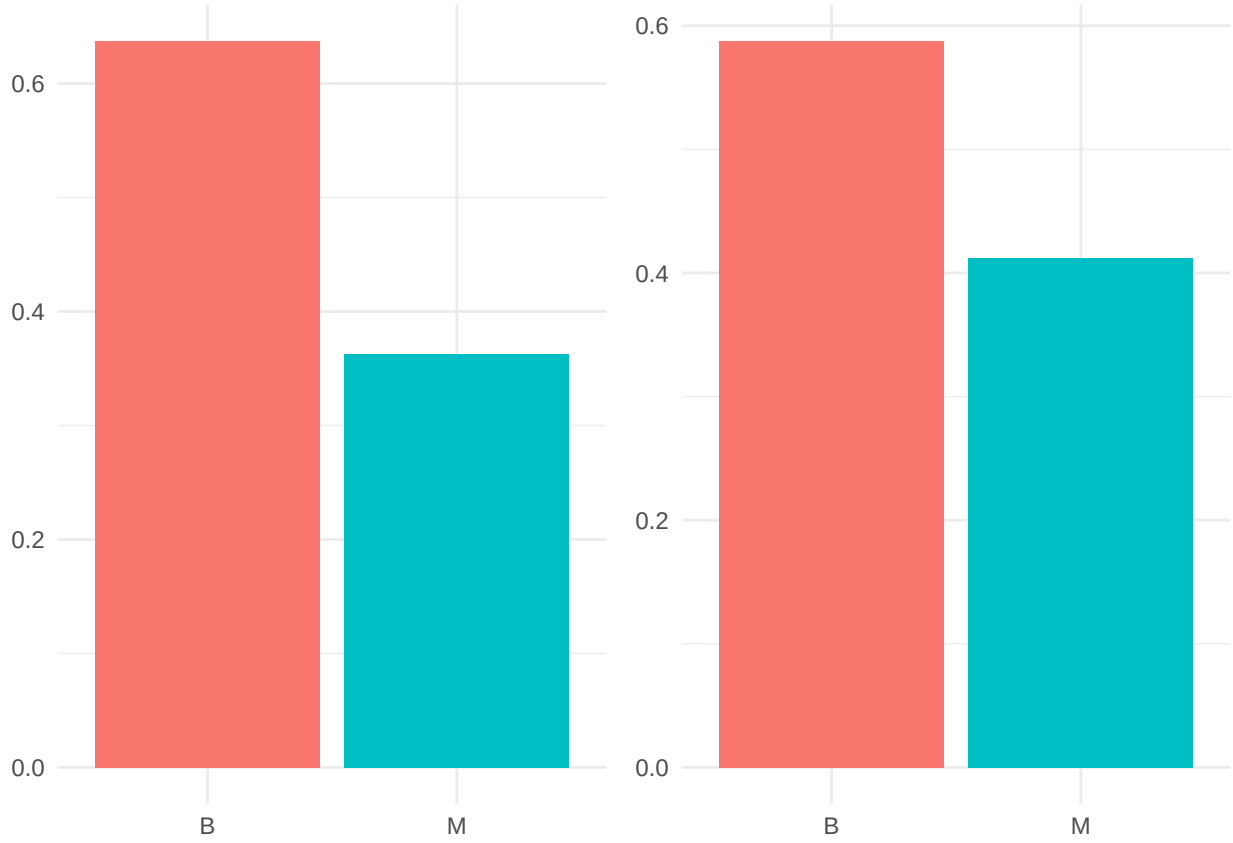
```
train <- data %>% sample_frac(0.8)
test <- anti_join(data, train, by = "id")

# need ids for later
id_train <- train$id
id_test <- test$id
```

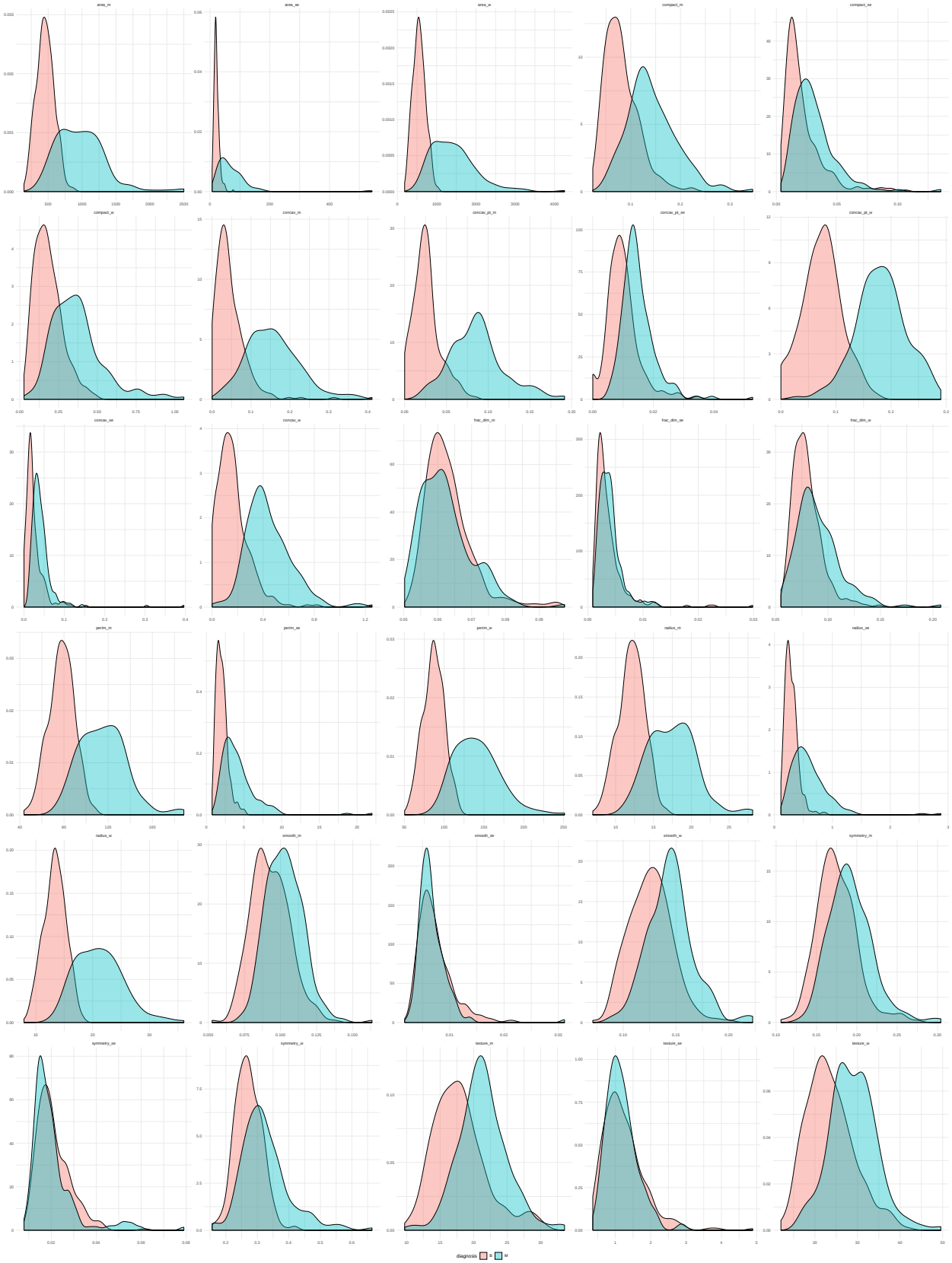
```
data %<>%
  dplyr::select(-c(id, X33))
train %<>%
  dplyr::select(-c(id, X33))
test %<>%
  dplyr::select(-c(id, X33))
```

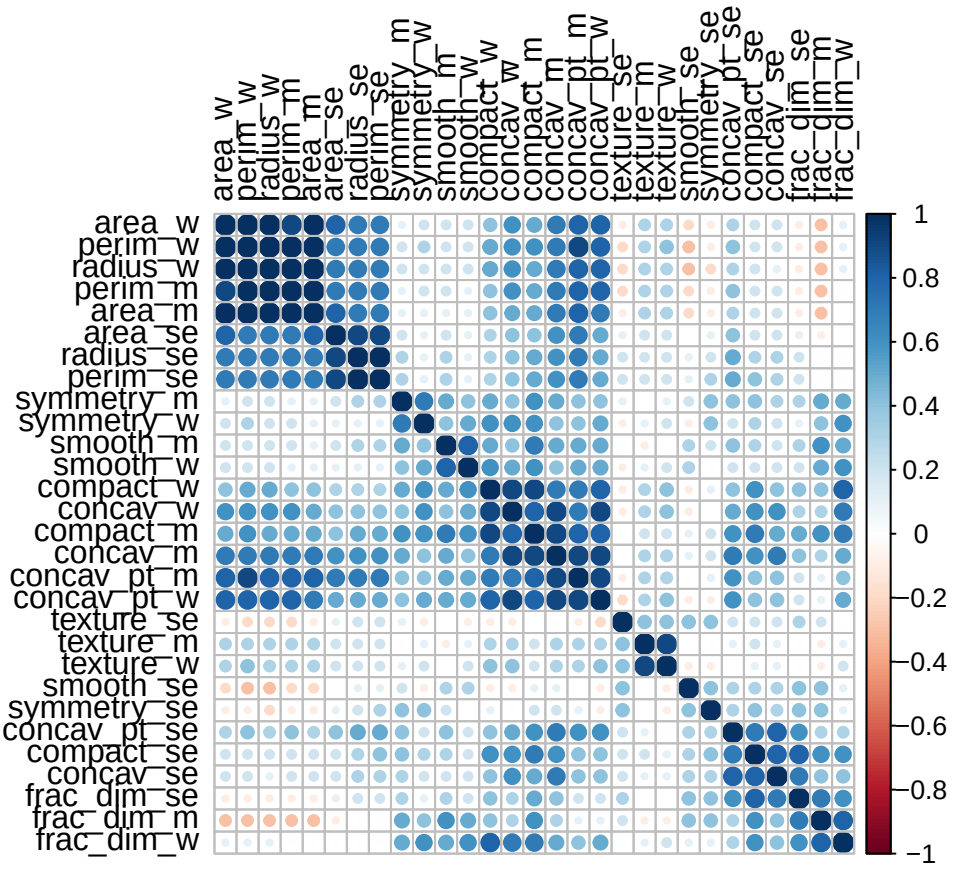
```
sum(is.na(data))
```

```
## [1] 0
```

```
training_data <- train[2:dim(train)[2]]
training_classes <- train[1]
test_data <- test[2:dim(test)[2]]
test_classes <- test[1]
```




area_m
area_se
area_w
compact_m
compact_se
compact_w
concav_m
concav_pt_m
concav_pt_se
concav_pt_w
concav_se
concav_w
frac_dim_m
frac_dim_se
frac_dim_w
perim_m
perim_se
perim_w
radius_m
radius_se
radius_w
smooth_m
smooth_se
smooth_w
symmetry_m
symmetry_se
symmetry_w
texture_m
texture_se
texture_w
diagnosis B M


area_w
perim_w
radius_w
perim_m
area_m
area_se
radius_se
perim_se
symmetry_m
symmetry_w
smooth_m
smooth_w
compact_w
concav_w
compact_m
concav_m
concav_pt_m
concav_pt_w
texture_se
texture_m
texture_w
smooth_se
symmetry_se
concav_pt_se
compact_se
concav_se
frac_dim_se
frac_dim_m
frac_dim_w
1
0.8
0.6
0.4
0.2
0
−0.2
−0.4
−0.6
−0.8
−1

Dimensionality Reduction and Feature Selection

PCA

For data of a non-trivial dimensionality it can be difficult to know where to begin with the modelling process as we may not have an intuitive idea of the underlying structure in our code. One such method of visualising the data in a lower space is principal component analysis (PCA). PCA aims to produce a set of linearly uncorrelated variables from our original set of variables of a reduced size. It does this by first taking the dataset represented as a matrix:

$$X = \begin{pmatrix} x_1^T \\ \vdots \\ x_2^T \end{pmatrix}$$

then we form a matrix normalised by the standard score by subtracting the column means from every column and dividing every column by the standard deviation for that column as in:

```
normalise_z <- function(X) {
  mean_cols <- colMeans(X)
  sd_cols <- apply(X, 2, sd)
  mean_normalised_X <- t(apply(X, 1, function(x) {
    x - mean_cols
  }))
  normalised_X <- t(apply(mean_normalised_X, 1, function(x) {
    x / sd_cols
  }))
  return(normalised_X)
}
```

Giving us:

$$Z = \begin{pmatrix} \frac{x_{11}-\mu_1}{\sigma_1} & \cdots & \frac{x_{1d}-\mu_d}{\sigma d} \\ \vdots & \vdots & \vdots \\ \frac{x_{n1}-\mu_1}{\sigma_1} & \cdots & \frac{x_{nd}-\mu_d}{\sigma d} \end{pmatrix}$$

Multiplying this by its transpose gives us the correlation matrix where the entry ρ_{ij} is the correlation between observation i and observation j. We can take the eigendecomposition of this matrix product to give us:

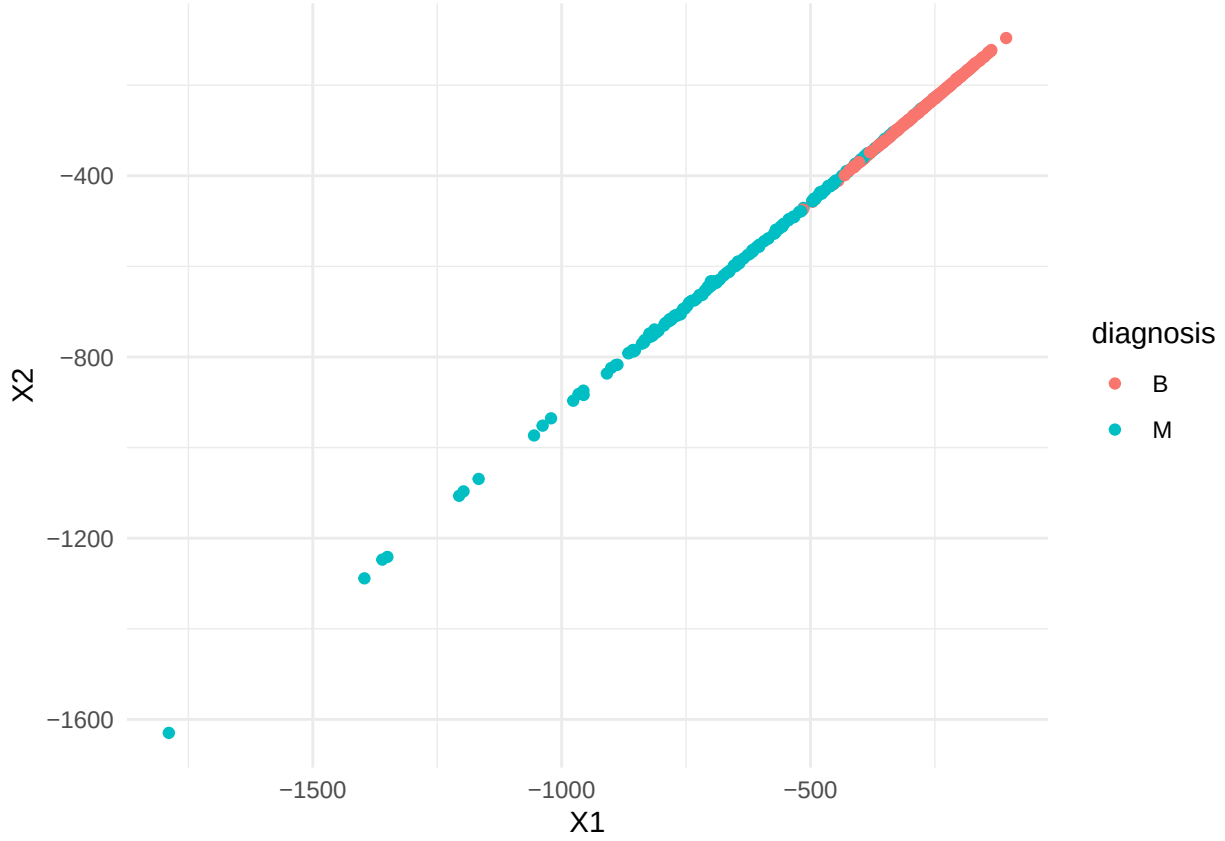
$$Z^T Z = P \Sigma^{-1} P^T$$

Where we assume that the diagonal Σ is ordered by size. The eigenvectors corresponding to the largest eigenvalues represent the combinations of features which account for the highest variance. If we wish to visualise at dimension $k < d$ we can simply take the top k eigenvectors as a matrix and multiply our data by this visualise our data in the reduced space. Code that does this can be found below.

```
pca <- function(X, number_components_keep) {
  normalised_X <- normalise_z(X)

  corr_mat <- t(normalised_X) %*% normalised_X

  eigenvectors <- eigen(corr_mat, symmetric = TRUE)$vectors

  reduced_data <- X %*% eigenvectors[, 1:number_components_keep]
  relevant_eigs <- eigenvectors[, 1:number_components_keep]
```

```
  returnds <- list(reduced_data, relevant_eigs)
  names(returnds) <- c("reduced_data", "reduction_matrix")
  return(returnds)
}
```

When applying this to the dataset and keeping 2 components the following plot results:



As we can see from this plot there is clearly some structure within the data that we can exploit for classification.

Classification

add list of methods+measure

SVM

From the application of PCA to the dataset we can see that, after reducing to 2 dimensions, the data appears to be almost linearly separable. Given this, an appropriate method of classifying the data would be to apply a soft-margin SVM to the reduced dimension data. Soft-margin SVMs solve the problem of classifying non-separable data by permitting a certain number of points to be incorrectly classifed however the number and the amount they violate the constraints by must be as small as possible. After manipulating the reformulated optimisation problem we end up with the optimisation problem

$$\min_{\lambda} \frac{\overline{\lambda} X X^T \overline{\lambda^T}}{4} + \lambda^T \mathbf{1}$$

$$\text{such that } 0 \leq \lambda_i \leq C$$

$$\text{and } \sum_{i}^{n} \lambda_i y_i = 0$$

where

$$X = \begin{pmatrix} x_1^T \\ \vdots \\ x_n^T \end{pmatrix} \text{ and } \overline{\lambda} = [\lambda_1 \cdot y_i, ..., \lambda_n \cdot y_n] \text{ and } \mathbf{1} = [1, ..., 1] \in \mathbb{R}^{\aleph}$$

For some predefined C. This is a quadratic programming problem with linear constraints which can be solved using the R package `quadprog`with the function `solve.QP`. From its documentation, this function can solve (for b) problems in the form $\min_b(-d^T b + \frac{1}{2} b^T D b)$ with the constraints that $A^T b \geq b_0$. By transforming the above problem into this format we can implement soft-margin SVM using the following code

```
train_soft_svm <- function(X, y, C) {

  num_observation <- nrow(X)
  dim_num <- ncol(X)

  Dmat2 <- diag(y) * X %*% t(X) %*% diag(y)
  diag(Dmat2) <- diag(Dmat2) + 1e-6
  dv2 <- rep(1, num_observation)

  A2 <- rbind( y,diag(num_observation))
  A2 <- rbind(A2, -1*diag(num_observation))

  bv2 <- c(c(0), rep(0, num_observation), rep(-C, num_observation) )
  model <- solve.QP(Dmat2, dv2, t(A2), bv2, meq = 1)
}
```

In order to recover w and b from λ we use the relationship

$$w = \sum_{i=0}^{n-1} \lambda_i x_i^T y_i$$

and

$$b = \text{mean}(\sum_{i=0}^{k} y_i - w \cdot x_i) \,.\, \forall\, i.0 < \lambda_i < C$$

Which can be made as functions in R as so:

```
calculate_b <- function(w, X, y, a, C) {
  ks <- sapply(a, function(x) {
    return(x > 0 && x < C)
  })
  indices <- which(ks)
  sum_bs <- 0
  for (i in indices) {
    sum_bs <- sum_bs + (y[i] - w %*% X[i, ])
  }
  return(sum_bs / length(indices))
}


recover_w <- function(a, y, X) {
  colSums(diag(a) %*% diag(y) %*% X)
}
```

From the parameters we can recover the equation of the line corresopnding the decision boundary which can later be used for plotting

```
soft_margin_svm_plotter <- function(w, b) {
  plotter <- function(x) {
    return(1 / w[2] * -(b + (w[1] * as.numeric(x))))
  }
  return(plotter)
}
```

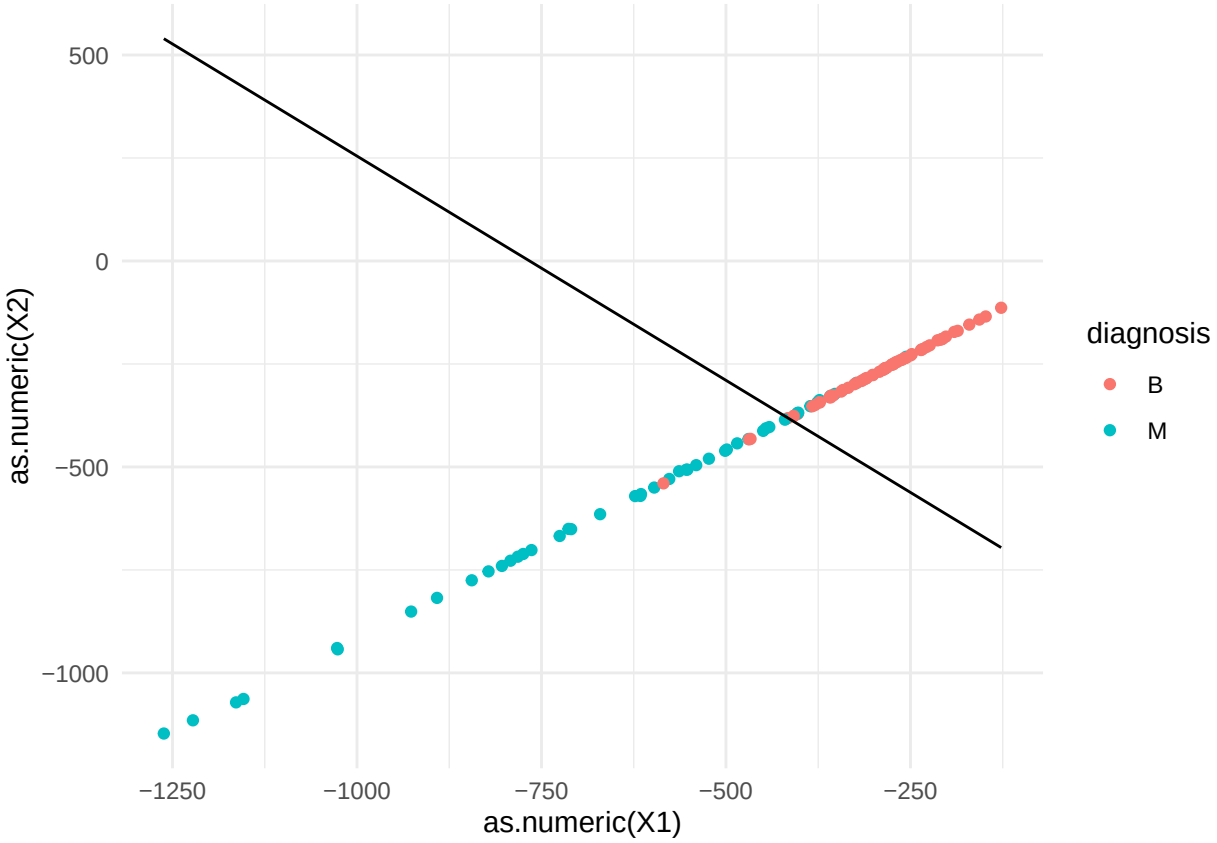
TODO: prediction function

We are now able to define a function which first embeds the vector in the reduced space and then predicts the class based upon the prediction function produced by the parameters found by the SVM. Below is code that will do the following based upon a package written for this assignment found here

```
model <- svm(
  X = pca_result$reduced_data,
  classes = numeric_training_labels,
  C = 100000, margin_type = "soft",
  kernel_function = linear_kernel,
  feature_map = linear_basis_function
)


reduced_prediction_fn <- model$prediction_function

pca_reduced_prediction_fn <- function(x) {
  p <- x %*% pca_result$reduction_matrix
  reduced_prediction_fn(t(p))
}
```

```
pca_reduced_test_data <- t(apply(as.matrix(test_data), 1, function(x) x %*%
                                 pca_result$reduction_matrix))
```

When running the trained SVM prediction functio on the test set we achieve 87.7192982 %. We can plot this code as below



Naive Bayes

Mathematical setting

Let y be the class label that we want to assign to an observation $\boldsymbol{x} = (x_1, \cdots x_d)$, where $x_1, \cdots x_d$ are the features. The probability of an observation having label y is given by Bayes rule,

$$\begin{aligned} P(y|x_1, \cdots, x_d) &= \frac{P(x_1, \cdots, x_d|y_k)P(y)}{P(x_1, \cdots, x_d)} \\ &\propto P(x_1, \cdots, x_d|y_k)P(y). \end{aligned}$$

The prior class probability $P(y)$ can be easily obtained by the proportion of observation that are in the given class.

The main assumption is that every feature is conditionally independent given the class label y. The reason why this classifier is called *naive* is that very often this assumption is not actually realistic.

This assumption simplifies the posterior to

$$P(y|x_1, \cdots, x_d) \propto P(y) \prod_{i=1}^{d} P(x_i|y).$$

There are various types of Naive Bayes classifiers based on the type of features. In our case, since we have continuous variables we assume that all features are normally distributed. Therefore, the conditional probabilities can be calculated as

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

Finally, to assign the class to an observation we use the Maximum A Posteriori decision rule. For every observation, we pick the class the has the highest probability

$$y = \underset{y}{\text{argmax}} P(y) \prod_{i=1}^{d} P(x_i|y).$$

Implementation

Here are some code snippets just to illustrate how these theoretical aspects are implemented. The full code can be found in the package.

The observations are stored as rows in X and the corresponding class labels are entires in the column matrix y.

First we calculate the prior class probabilities based on the number of observations in each class.

```
n <- dim(X)[1]
d <- dim(X)[2]
classes <- sort(unique(y)[, 1])
k <- length(classes)

prior <- rep(0, k)
for (i in 1:k) {
  prior[i] <- sum(y == classes[i]) / n
}
```

Then we create an array of the mean and sd of the data split by clasess and features.

```
summaries <- array(rep(1, d * k * 2), dim = c(k, d, 2))
for (i in 1:k) {
  X_k <- X[which(y == (i - 1)), ]
  summaries[i, , 1] <- apply(X_k, 2, mean)
  summaries[i, , 2] <- apply(X_k, 2, sd)
}
```

Finally, the predictions are obtained by taking the largest posterior class probability. Note that in order to avoid underflow, we take the maximum of the *log* posterior class probabilities.

```
probs <- matrix(rep(0, n * k), nrow = n)
for (obs in 1:n) {
  for (class in 1:k) {
    class_prob <- log(prior[class])
    for (feat in 1:d) {
      mu <- summaries[class, feat, 1]
      sd <- summaries[class, feat, 2]
      cond <- dnorm(x_new[obs, feat], mu, sd, log = TRUE)
      class_prob <- class_prob + cond
    }
    probs[obs, class] <- class_prob
  }
}

pred <- apply(probs, 1, which.max)
```

Fit model to dataset

```
levels(training_classes$diagnosis) <- c(0, 1)
training_classes %<>% as.matrix
mode(training_classes) <- "numeric"

levels(test_classes$diagnosis) <- c(0, 1)
test_classes %<>% as.matrix
mode(test_classes) <- "numeric"
```

Fit the Naive Bayes model to the data, calculate predictions and check the accuracy using.

```
model_naive <- naive_bayes(training_data, training_classes)

pred_naive <- predict(model_naive, as.matrix(test_data))

# confusion_plot(test_classes,pred_naive)

calc_accuracy <- function(ytest, yhat) sum(drop(yhat) == drop(ytest)) / length(drop(ytest))

acc_naive <- calc_accuracy(test_classes, pred_naive)
acc_naive
```

```
## [1] 0.9122807
```

Fit the model to the PCA-reduced dataset.

```
model_naive_pca <- naive_bayes(pca_result$reduced_data, training_classes)

pred_naive_pca <- predict(model_naive_pca, pca_reduced_test_data)

acc_naive_pca <- calc_accuracy(test_classes, pred_naive_pca)
acc_naive
```

```
## [1] 0.9122807
```

Logistic Regression

Mathematical Setting

Let $Y_i \mid \mathbf{x}_i \sim \text{Bernoulli}(p_i)$ with $p_i = \sigma(\mathbf{x}_i^\top \boldsymbol{\beta})$ where $\sigma(\cdot)$ is the **sigmoid function**. The joint log-likelihood is given by

$$\ln p(\mathbf{y} \mid \boldsymbol{\beta}) = \sum_{i=1}^{n} y_i \ln(p_i) + (1 - y_i) \ln(1 - p_i) = -\sum_{i=1}^{n} \ln\left(1 + \exp((1 - 2y_i)\mathbf{x}_i^\top \boldsymbol{\beta})\right)$$

Maximum Likelihood Estimation

Maximizing the likelihood is equivalent to minimizing the negative log-likelihood. Minimizing the negative log likelihood is equivalent to solving the following optimization problem

$$\min_{\boldsymbol{\beta}} \sum_{i=1}^{n} \ln\left(1 + \exp((1 - 2y_i)\mathbf{x}_i^\top \boldsymbol{\beta})\right)$$

Maximum-A-Posteriori and Ridge Regularization

We can introduce an isotropic Gaussian prior on **all** the coefficients $p(\boldsymbol{\beta}) = N(\mathbf{0}, \sigma_{\boldsymbol{\beta}}^2 I)$. Maximizing the posterior $p(\boldsymbol{\beta} \mid \mathbf{y})$ is equivalent to minimizing the negative log posterior $-\ln p(\boldsymbol{\beta} \mid \mathbf{y})$ giving

$$\min_{\boldsymbol{\beta}} \sigma_{\boldsymbol{\beta}}^2 \sum_{i=1}^{n} \ln\left(1 + \exp((1 - 2y_i)\mathbf{x}_i^\top \boldsymbol{\beta})\right) + \frac{1}{2}\boldsymbol{\beta}^\top \boldsymbol{\beta}$$

Gradient Ascent

Maximum Likelihood updates take the form

$$\boldsymbol{\beta}_{k+1} \leftarrow \boldsymbol{\beta}_k + \gamma X^\top (\mathbf{y} - \sigma(X\boldsymbol{\beta}_k))$$

wheras for MAP we have

$$\boldsymbol{\beta}_{k+1} \leftarrow \boldsymbol{\beta}_k + \gamma_k \left[\sigma_{\boldsymbol{\beta}}^2 X^\top (\mathbf{y} - \sigma(X\boldsymbol{\beta}_k)) - \boldsymbol{\beta}_k\right]$$

Newton's Method

For stability, we add a learning rate, which is in practice often set to $\alpha = 0.1$. The iterations for Maximum Likelihood take the form of

$$\boldsymbol{\beta}_{k+1} \leftarrow \boldsymbol{\beta}_k + \alpha (X^\top D X)^{-1} X^\top (\mathbf{y} - \sigma(X\boldsymbol{\beta}_k))$$

whereas for MAP take the form

$$\boldsymbol{\beta}_{k+1} \leftarrow \boldsymbol{\beta}_k + \alpha \left[\sigma_{\boldsymbol{\beta}}^2 X^\top D X + I\right]^{-1} \left(\sigma_{\boldsymbol{\beta}}^2 X^\top (\mathbf{y} - \sigma(X\boldsymbol{\beta}_k)) - \boldsymbol{\beta}_k\right)$$

In practice, for example in MLE, we would solve the corresponding system for $\mathbf{d}_k$

$$(X^\top D X)\mathbf{d}_k = \alpha X^\top (\mathbf{y} - \sigma(X\boldsymbol{\beta}_k))$$

and then perform the update

$$\boldsymbol{\beta}_{k+1} \leftarrow \boldsymbol{\beta}_k + \mathbf{d}_k$$

Implementation and Results

Notice that differently from the methods used above, Logistic Regression requires our data to have a constant feature of 1s in order to fit the bias coefficient.

We've implemented from scratch Newton's Method and Gradient Ascent for both Maximum Likelihood Estimation and Maximum-A-Posteriori, with the formulas shown above. In addition, we've used the BFGS implementation provided by the R `optim` function.

Below we can see the accuracy of the different optimization methods (BFGS, Newton's method and Gradient Ascent) on the different cost functions (MAP or MLE).

	MLE	MAP
BFGS	0.9561404	0.9385965
NM	0.9298246	0.9298246
GA	0.9298246	0.8771930

Random-Walk Metropolis-Hastings Implementation on Reduced Data

Similarly as to what we've done with SVMs and Naive Bayes, we can apply Logistic Regression to the data after it has been reduced to two dimensions using PCA. Since we will be working in two dimensions, this allow us to plot the results more neatly in 2D and then sample from the (unnormalized) log posterior for Logistic Regression and therefore have a measure of uncertainty around our decision boundary. In particular, we can define a function that performs Random Walk Metropolis Hastings with a normal proposal distribution. To make it more efficient we can work with the log posterior and change the decision rule. I In addition, we can pre-calculate all the normal and uniform samples and just access them later. The full algorithm that we will be using is as follows

Algorithm 1: Metropolis-Hastings

Set starting value $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}_{\text{MAP}}$.
Sample from standard MVNs $\mathbf{n}_1, \ldots, \mathbf{n}_N \sim N(\mathbf{0}, H^{-1}(\boldsymbol{\beta}_{MAP}))$.
Sample from Uniform and take the log $u'_1, \ldots, u'_N \sim \mathcal{U}(0,1)$ and $u_i := \log(u'_i)$.
for $i = 1, 2, \ldots, N$ **do**:
    Draw a sample from the proposal distribution $\boldsymbol{\beta}^*_i \leftarrow \boldsymbol{\beta}_i + \mathbf{n}_i$
    if $u_i \leq \log p(\boldsymbol{\beta}^*_i) - \log p(\boldsymbol{\beta}_i)$:
        Accept candidate $\boldsymbol{\beta}_i \leftarrow \boldsymbol{\beta}^*_i$
    else:
        Reject $\boldsymbol{\beta}^*_i$ and use the value of $\boldsymbol{\beta}_i$ as the realization of $\boldsymbol{\beta}_{i+1}$.
    end
end

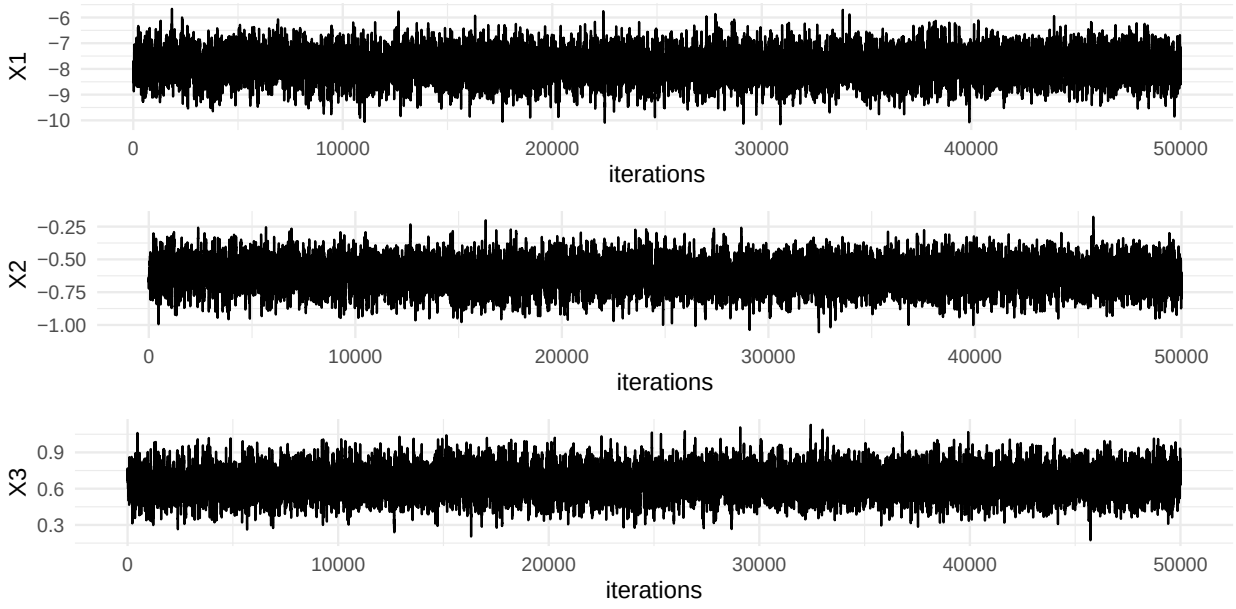
```
rwmh_multivariate_log <- function(start, niter, logtarget, vcov, thinning, burnin){
    # Set current z to the initial point and calculate its log target to save computations
    z  <- start; pz <- logtarget(start)
    # create vector deciding iterations where we record the samples
    store <- seq(from=(1+burnin), to=niter, by=thinning)
    # Generate matrix containing samples. Initialize with the starting value
    samples <- matrix(0, nrow=length(store), ncol=nrow(start))
    samples[1, ] <- start
    # Generate uniform random numbers in advance, to save computation. Log them.
    log_u <- log(runif(niter))
    # Proposal is a standard MVN. Generate samples and use linearity later
```

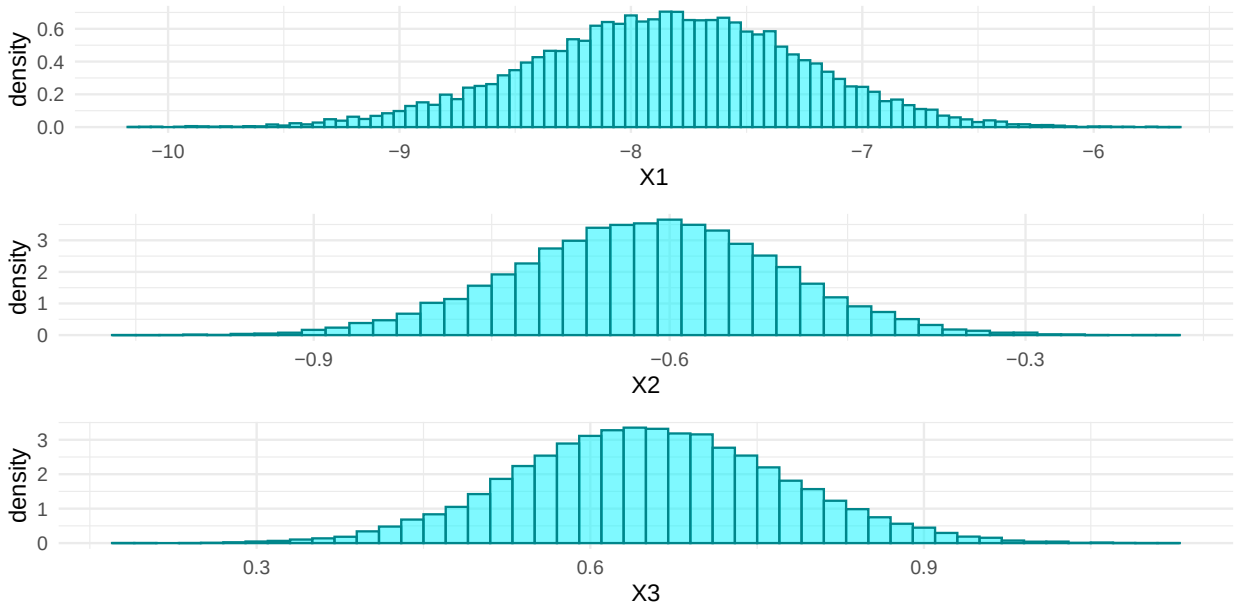
```
    vcov <- diag(nrow(start)) %*% vcov
    normal_shift <- mvrnorm(n=niter, mu=c(0,0,0), Sigma=vcov)
    for (i in 2:niter){
        # Sample a candidate and calculate log density there
        candidate <- z + normal_shift[i, ]
        p_candidate <- logtarget(candidate)
        if (log_u[i] <= p_candidate - pz){ # Modify decision rule with bijection
            z  <- candidate; pz <- p_candidate
        }
        # Finally add the sample to our matrix of samples
        if (i %in% store) samples[which(store==i), ] <- z
    }
  return(samples)
}
```

We start our RWMH using the MAP estimate and we use the inverse of the approximated hessian matrix (returned by `optim`) as the variance-covariance matrix for the proposal.

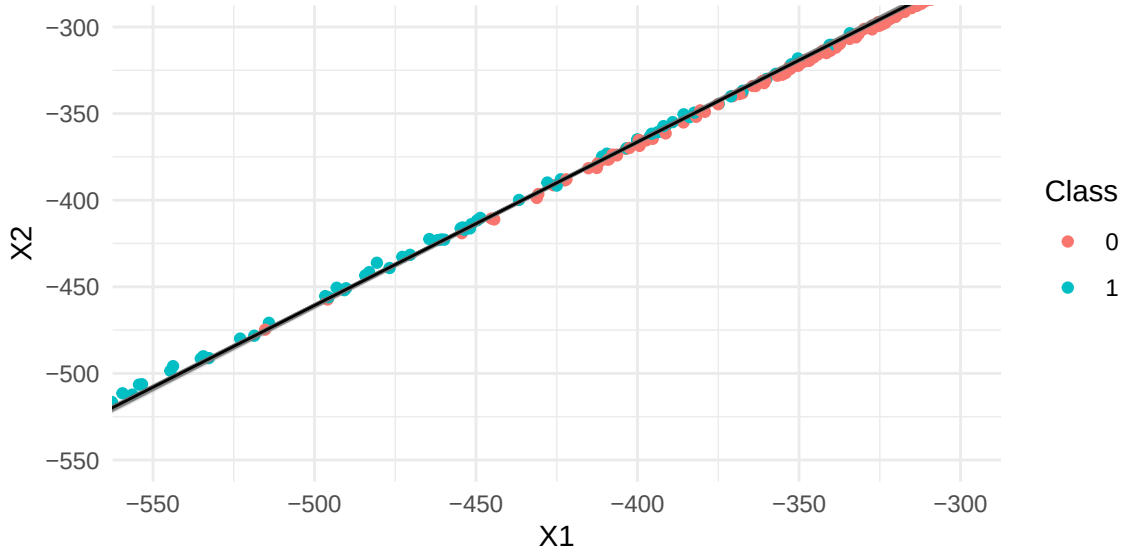
	MLE	MAP
BFGS	0.9210526	0.9122807
NM	0.9210526	0.9210526
GA	0.5877193	0.4122807

From the trace plots we notice the chain is mixing quickly, signalling a good exploratory behavior. The histograms of each coordinate of the samples show RWMH is sampling sensibly.





The next thing to do is to plot the decision boundary found by BFGS together with the sampled lines.



Conclusion

Merge all predictions

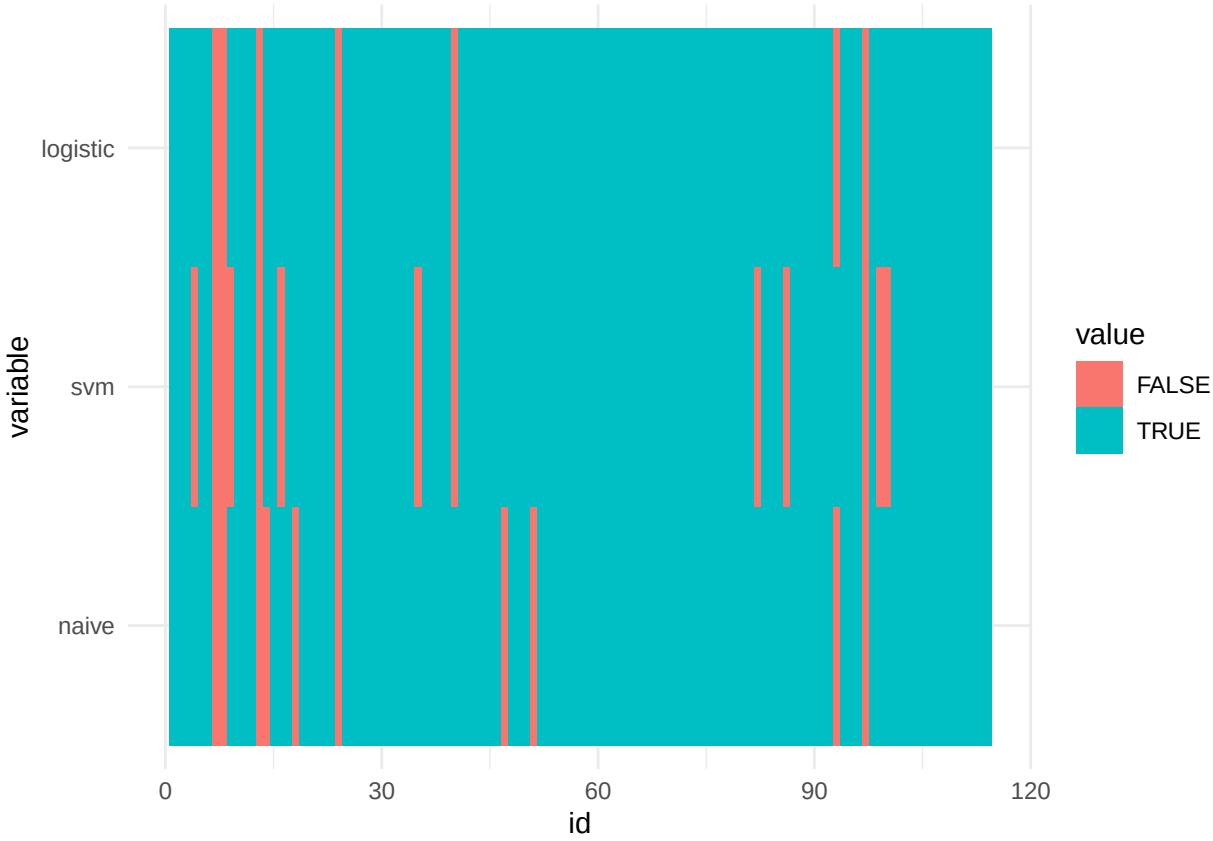
```
id <- seq(length(id_test))
all_pred <- cbind(id, numeric_test_labels, pred_naive, pred_svm)
colnames(all_pred) <- c("id", "actual", "naive", "svm")
all_pred[all_pred == -1] <- 0
all_pred %<>% as.data.frame()
```

```
confusion_plot <- function(actual, predicted) {
  confusion_matrix <- as.data.frame(table(actual, predicted))
  g <- ggplot(confusion_matrix, aes(x = actual, y = predicted)) +
    geom_tile(aes(fill = Freq)) +
    geom_text(aes(label = sprintf("%1.0f", Freq)), color = "white", fontface = "bold") +
    labs(x = "Actual class", y = "Predicted class") +
    theme_minimal()
  return(g)
}
```

```
errors <- all_pred %>%
  mutate(
    naive = naive == actual,
    svm = svm == actual,
    logistic = yhat_map_bfgs == actual
  ) %>%
  dplyr::select(-"actual") %>%
  melt(id = "id")
```

Visualize which are the observations that the models missclassify.

```
ggplot(errors, aes(x = id, y = variable, fill = value)) +
  geom_raster() +
  theme_minimal()
```


logistic
svm
naive
variable
0
30
60
90
120
id
value
FALSE
TRUE