Breast Cancer Project

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. The features describe characterstics of the cell nucleus wich three different statistics computed for each feature. The three statistics are the largest value computed of the feature computed for each image indicated with a "_w" appended to the feature name, the mean value of the feature for the image indicated with a "_m" appended, and the standard error. The features computed are the radius of the nuclei, the texture of the nuclei, area, perimeter, smoothness, compactness, concavity, concave points, symmetry, and fractal dimension.

Exploratory analysis

We begin by exploring the data to gain some understanding of the features and their relationships to inform further work. We do the majority of the data manipulation and analysis within this report using the tidyverse library. We begin by loading the data as a tibble:

```r
data <- read_csv("../data/data.csv")
```

After renaming some variables to some more manageable variables we next have to ensure the hygiene of our data. Through manual inspection it is clear that our data is already in the tidy data format so all that remains is to see if any corrupted data is present:

```r
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
```

From this we can see that the feature parsed as `X33` is not present for any observation and therefore can be removed without any difficulties using `dplyr`

```
data %<>%
  dplyr::select(-c(X33))
```

In order to make the data easier to work with we also need to set the response, the diagnosis of the patient, to be a factor.

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

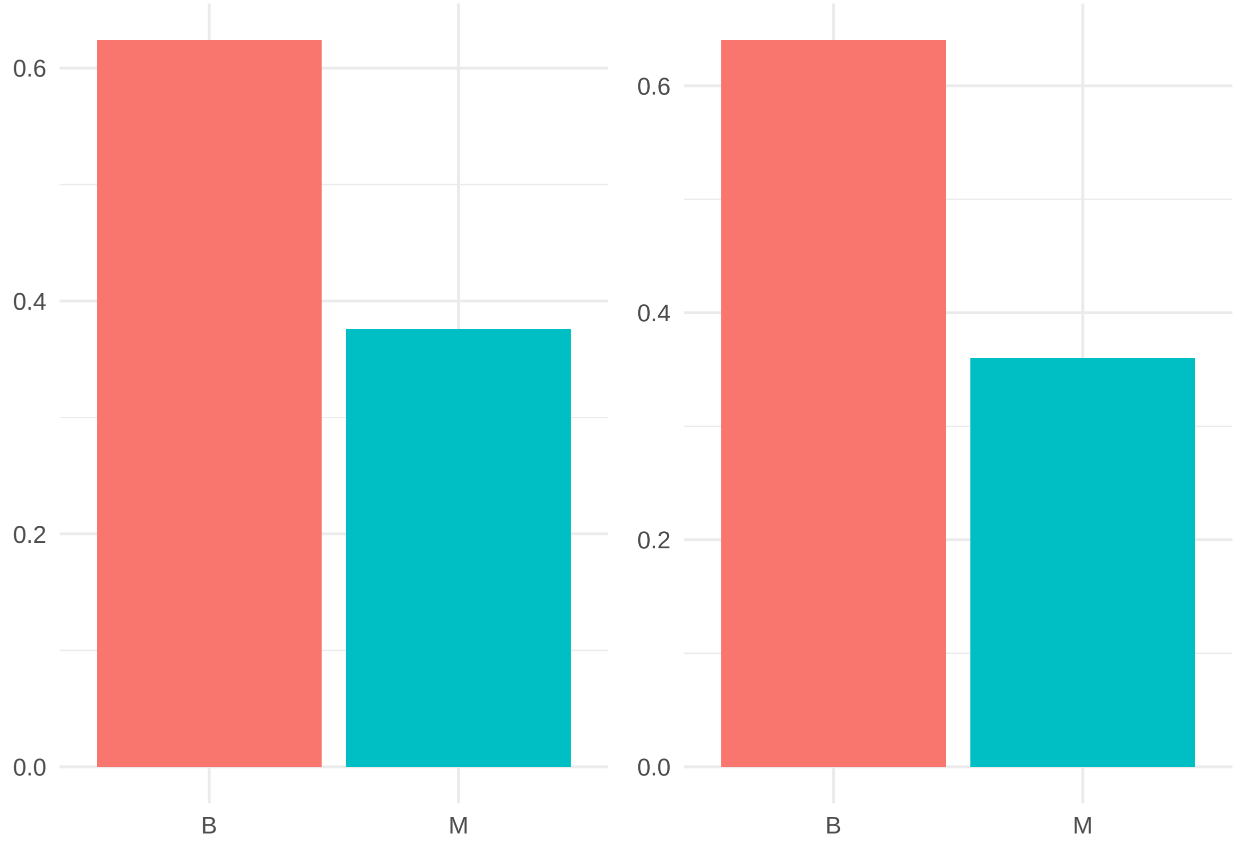
The next step before analysing the data is to produce a test/train split in order to evaluate how well our models and methods will generalise which we can do again using `dplyr`. We choose an 80/20 train/test split.

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

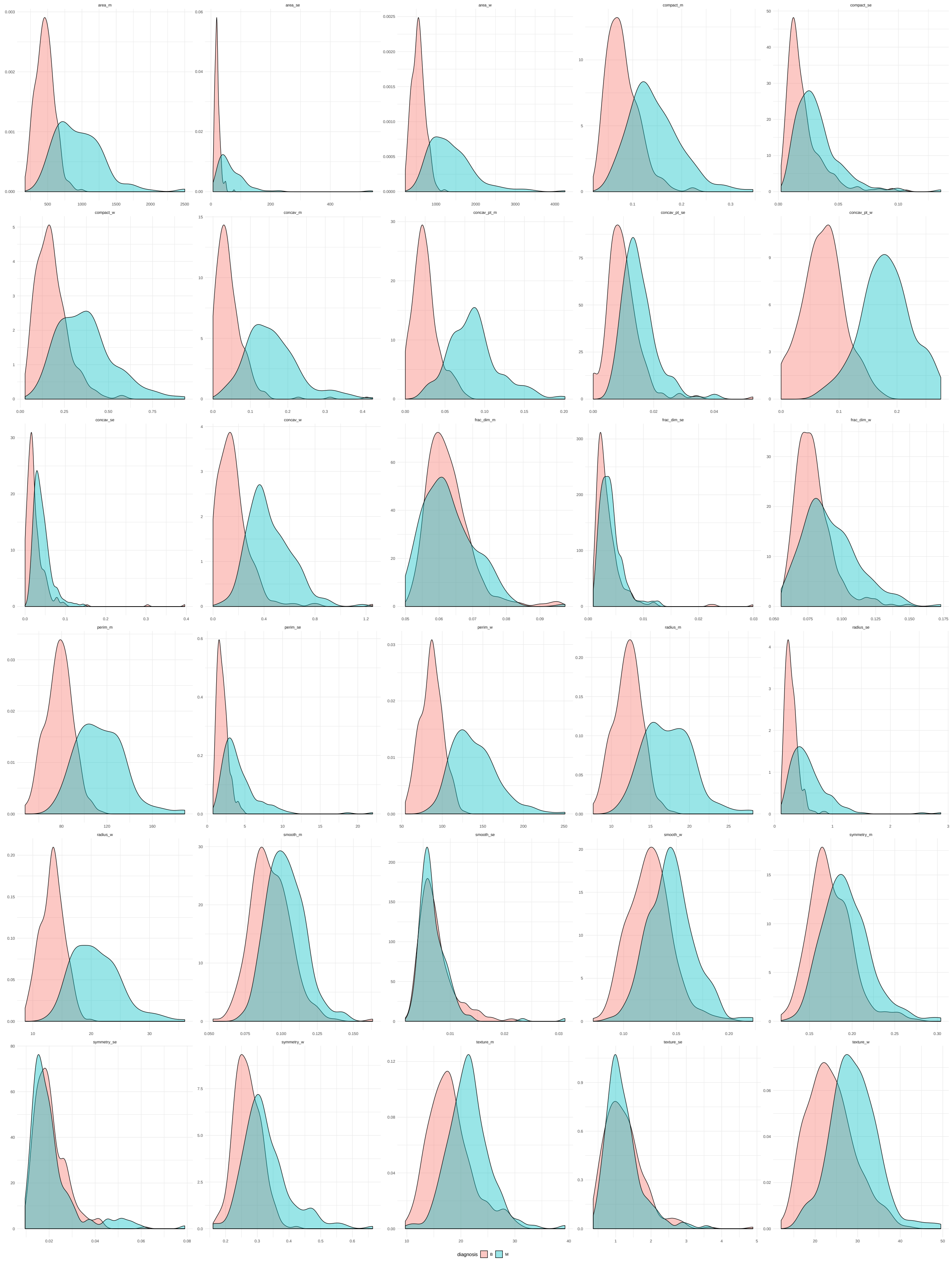
We also remove the labels and store them elsewhere so that we have the raw data input to analyse.

```
id_train <- train$id
id_test <- test$id

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

Before continuing with our analysis we need to make sure that the train/test split is representative of the entire data. Below is a plot of the class distribution in the training set on the left and the test set on the right. As can be seen these are fairly even and therefore this split is suitable for analysis.



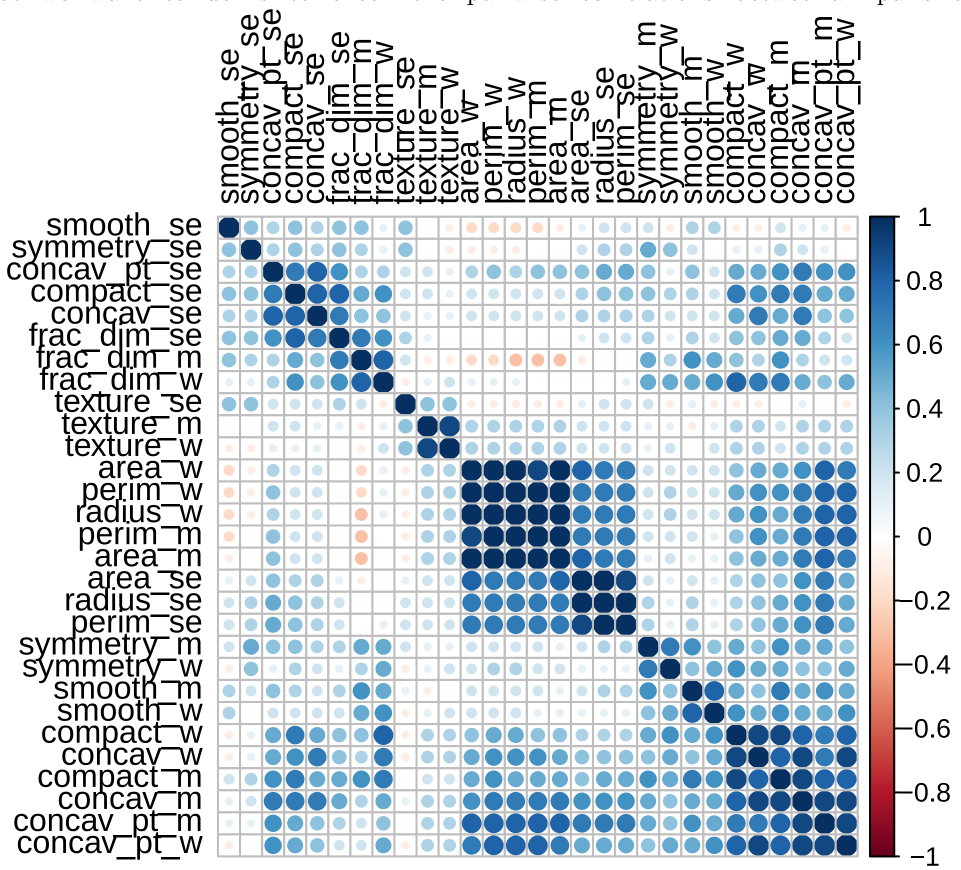
We now want to try and discover if our data has any exploitable structure that we can use for classification. We can examine the density plots of each feature for each class against each other to see if there are features which have substantially different plots.



From these plots above we can see that many of the features are, from visual, inspection, approximately normal. Whilst this is not conclusive as per a test however this may justify some techniques for classification which rely on normality of the features (as in the choice of naive bayes below).

Many of the features display little difference in densities between classes however several of the features display noticeable difference which is promising for modelling.

The final plot we want to do is to check the pairwise correlations betwee all pairs of features.



This plot shows that a large number of our features are highly correlated which gives us some indication that we can perhaps predict some features from others and reduce the dimension of the data we are working with. Now that we have established that there appears to be some structure within our data and that there appears to be more important variables we can move on with an attempt to reduce the dimensionality of our data.

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:

```r
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:

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

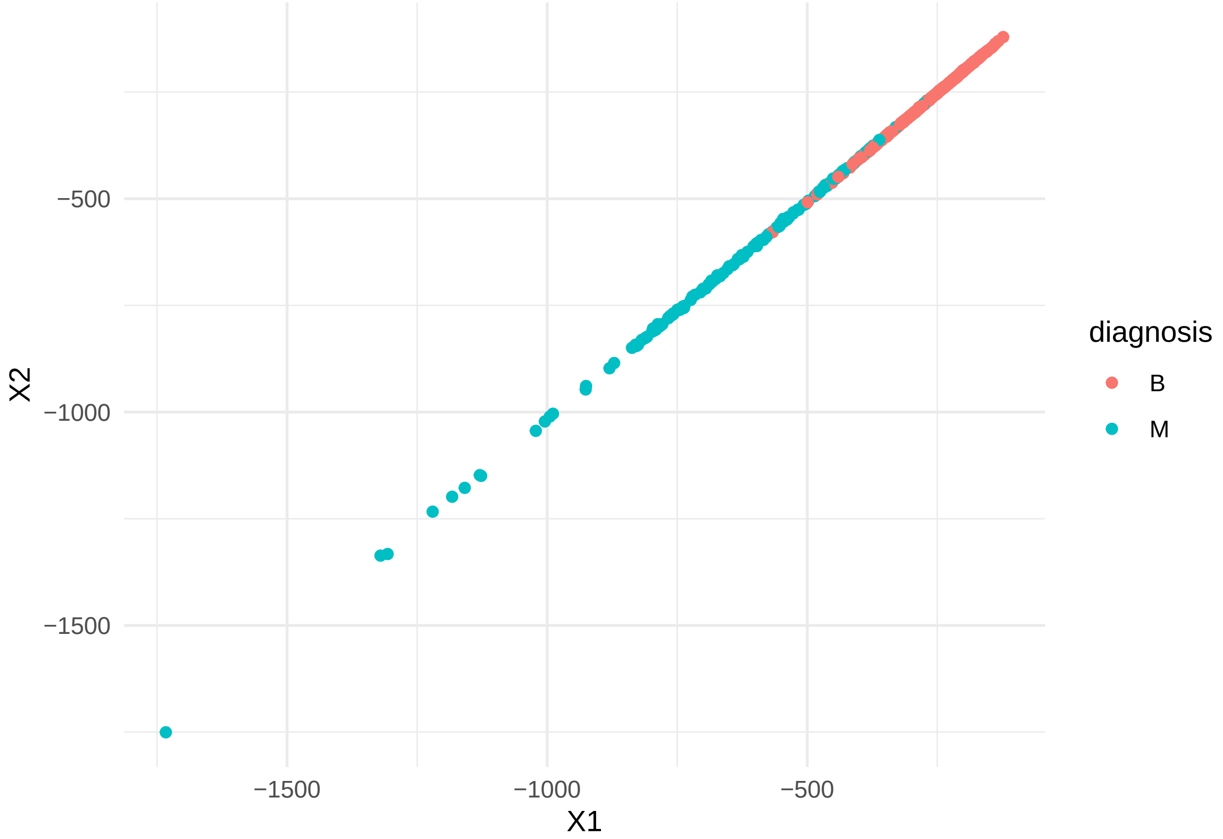
Multiplying this by its transpose gives us the correlation matrix where the entry ρ_{ij} is the correlation between observation i and observation j. We can take the eigendecomposition of this matrix product to give us:

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

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.

```r
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. This reduced dataset and transformation matrix will be used in classification attempts in the next section.

Classification

In this section we will explore various approaches of classifying the data. This includes using SVMs, Naive Bayes, and frequentist and bayesian approaches to logistic regression. We begin with SVMs applied to the reduced dataset.

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

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

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

```r
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)
}
```

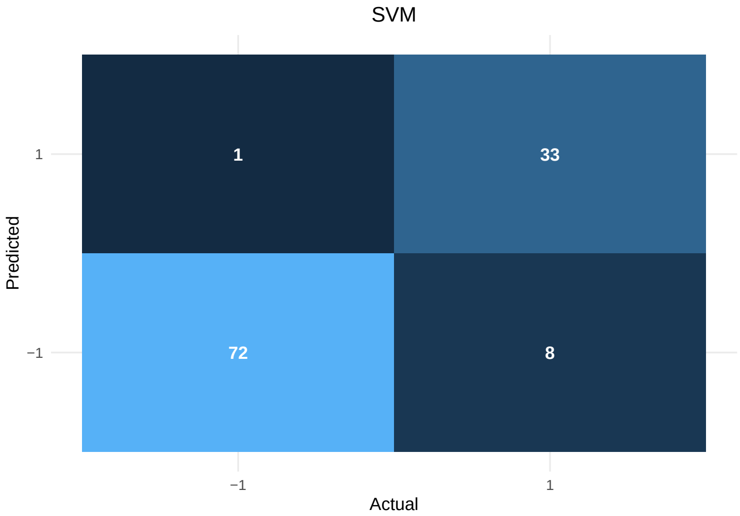
TODO: prediction function

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

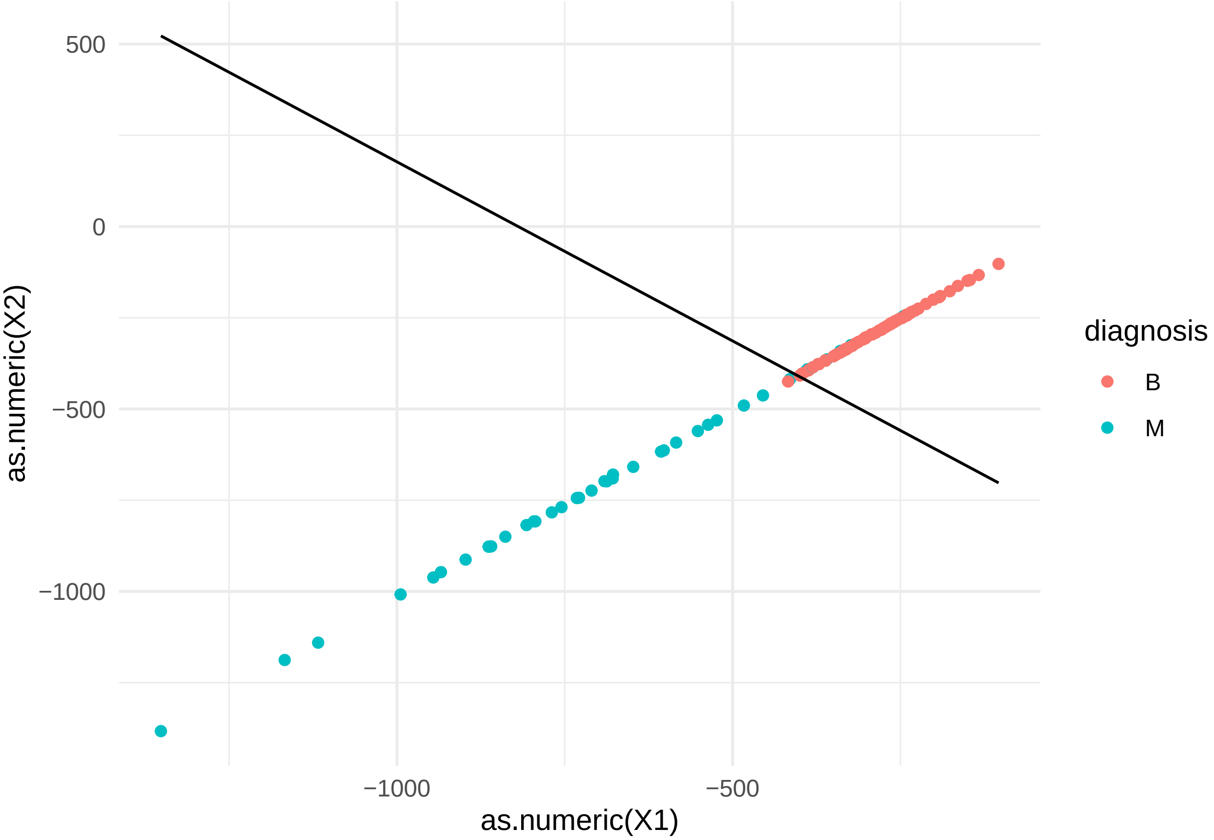
```
model <- svm(
  X = pca_result$reduced_data,
  classes = numeric_training_labels,
  C = 100000, margin_type = "soft",
  kernel_function = linear_kernel,
  feature_map = linear_basis_function
)


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))
```

```
confusion_plot(numeric_test_labels,pred_svm,'SVM')
```



When running the trained SVM prediction function on the test set we achieve 92.1052632 %. 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)
}
```

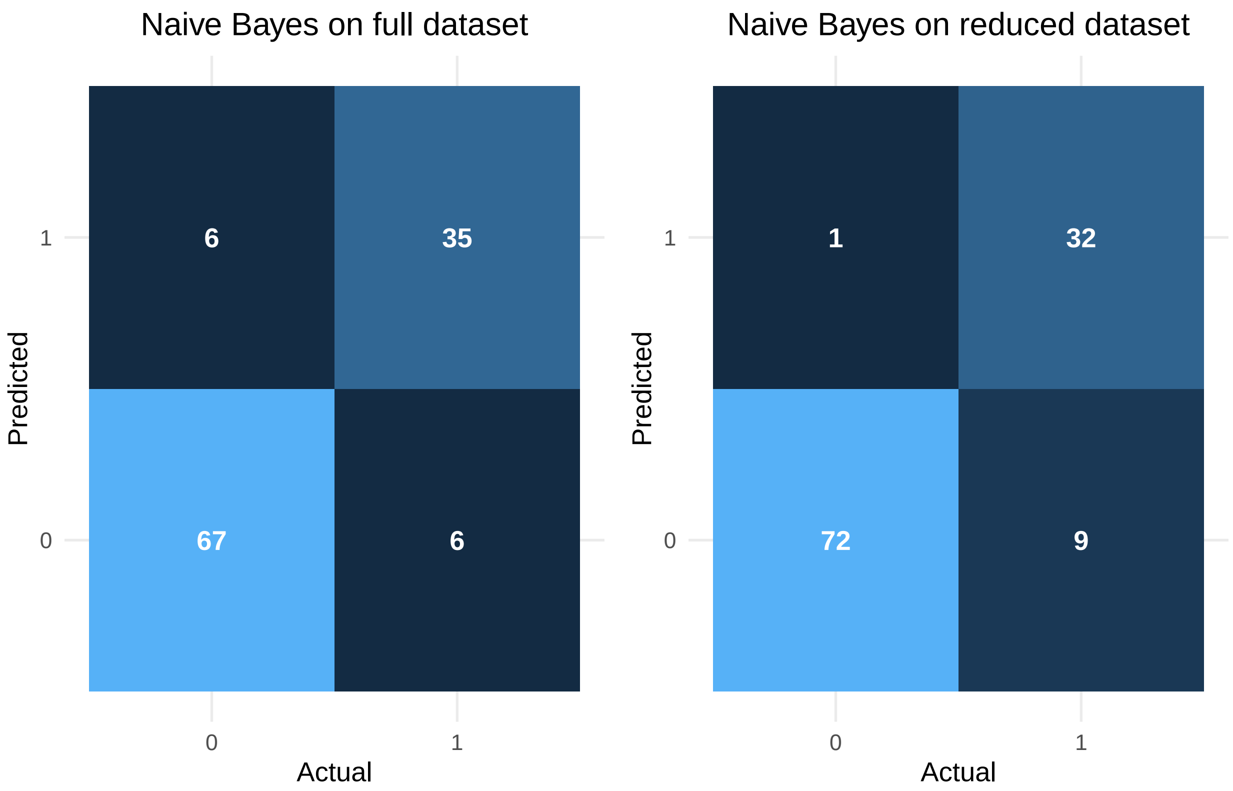
Finally, the predictions are obtained by taking the largest posterior class probability. Note that in order to avoid underflow, we take the maximum of the *log* posterior class probabilities.

```
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

Fitting the model to the full dataset and the reduced dataset we obtain the following accuracies.

```
g1 <- confusion_plot(test_classes,pred_naive,'Naive Bayes on full dataset')
g2 <- confusion_plot(test_classes,pred_naive_pca,'Naive Bayes on reduced dataset')
ggarrange(g1,g2)
```



full	0.8947
reduced	0.9123

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

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

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

The cost functions for maximum likelihood estimation and for maximum a posteriori can be implemented as follows.

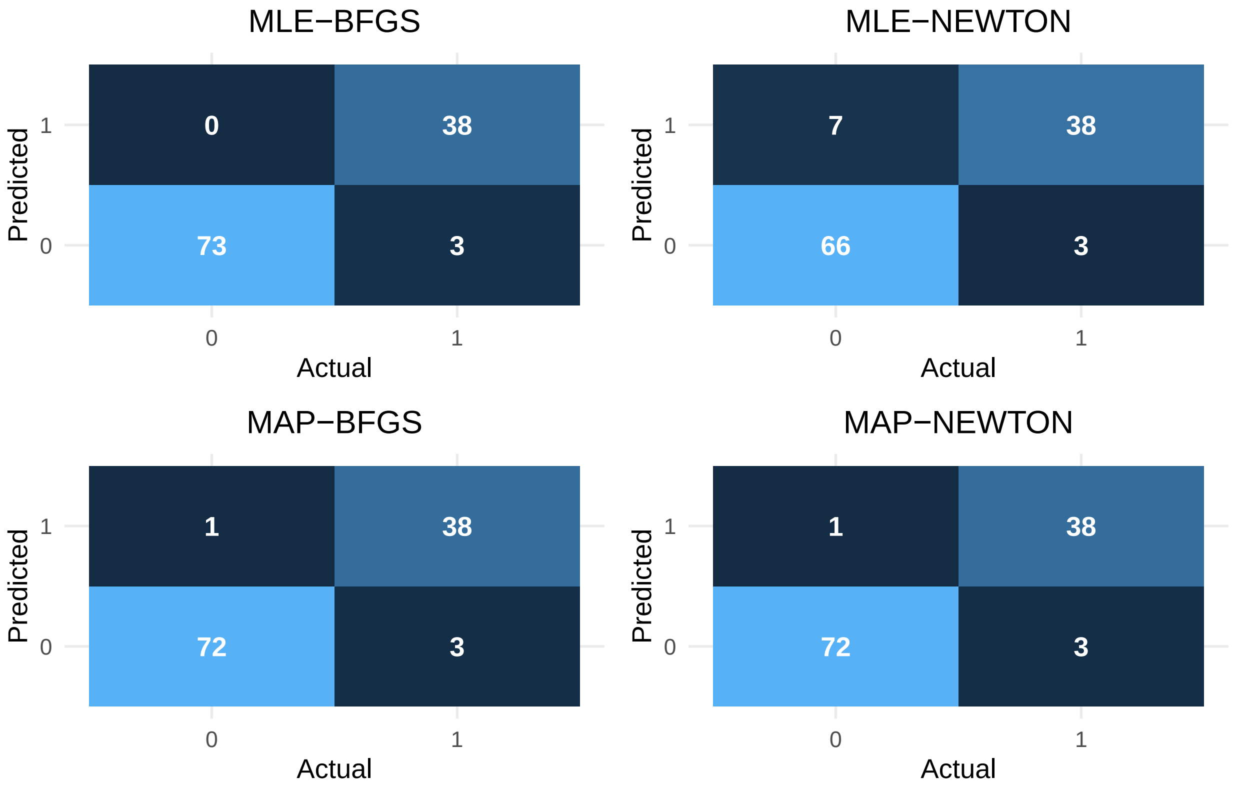
```
mle_cost <- function(beta) sum(log(1 + exp((1 - 2*y) * (X %*% beta))))
map_cost <- function(beta) (sigmab^2)*mle_cost(beta) + 0.5*sum(beta^2)
```

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 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 and Newton's method) on the different cost functions (MAP or MLE).

	MLE	MAP
BFGS	0.9736842	0.9649123
NM	0.9122807	0.9649123



Both MAP and MLE fail to provide us with uncertainty estimates. A fully-Bayesian approach is intractable due to the form of the joint likelihood but we can obtain samples from the posterior using a Random-Walk Metropolis Hastings algorithm. These samples will be parameter vectors defining different decision boundaries. In the following section we sample from the logistic posterior where the dataset has been embedded in two dimensions using PCA. This allows us to plot the decision boundaries given by the samples and thus have an idea of the uncertainty described by the posterior distribution.

Random-Walk Metropolis-Hastings Implementation on Reduced Data

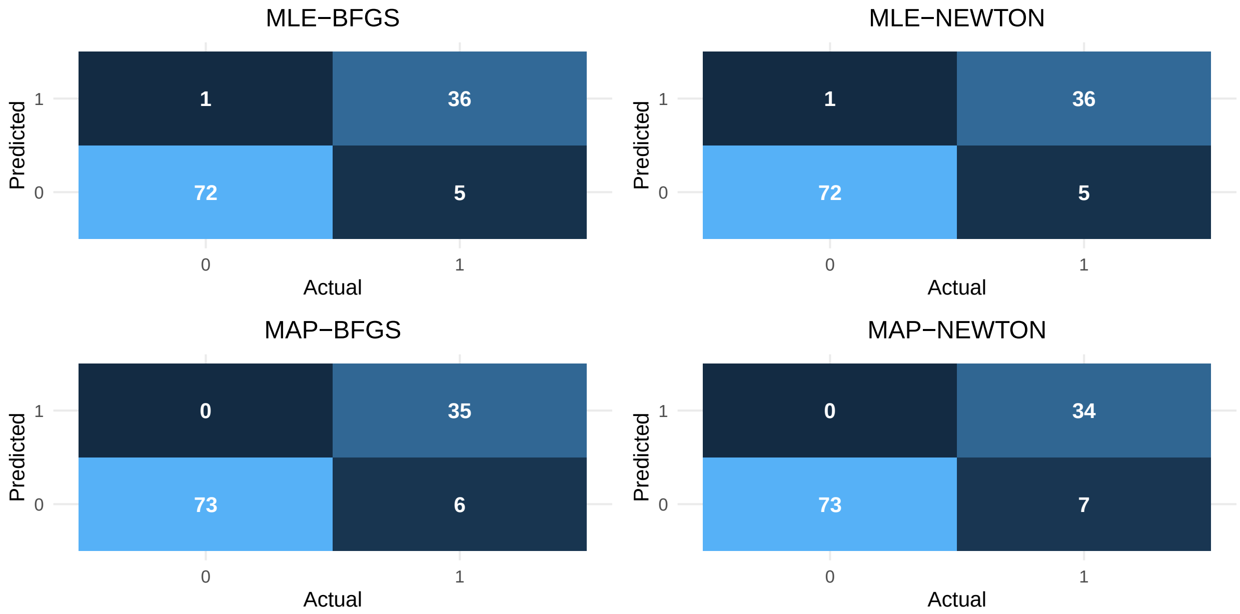
Similarly as to what we've done with SVMs and Naive Bayes, we can apply Logistic Regression to the data after it has been reduced to two dimensions using PCA. Below we implement a function for the Random-Walk Metropolis-Hastings algorithm and, to make it more efficient, we work with the log posterior, modify the decision rule and pre-calculate all the normal and uniform samples.

```
Algorithm 1: Metropolis-Hastings
Set starting value $\boldsymbol{\beta}_0 \leftarrow \boldsymbol{\beta}_{\text{MAP}}$.
Sample from standard MVNs $\mathbf{n}_1, \dots, \mathbf{n}_N \sim N(\mathbf{0}, H^{-1}(\boldsymbol{\beta}_{MAP}))$.
Sample from Uniform and take the log $u'_1, \dots, u'_N \sim \mathcal{U}(0,1)$ and $u_i := \log(u'_i)$.
for $i = 1, 2, \dots, N$ 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
```

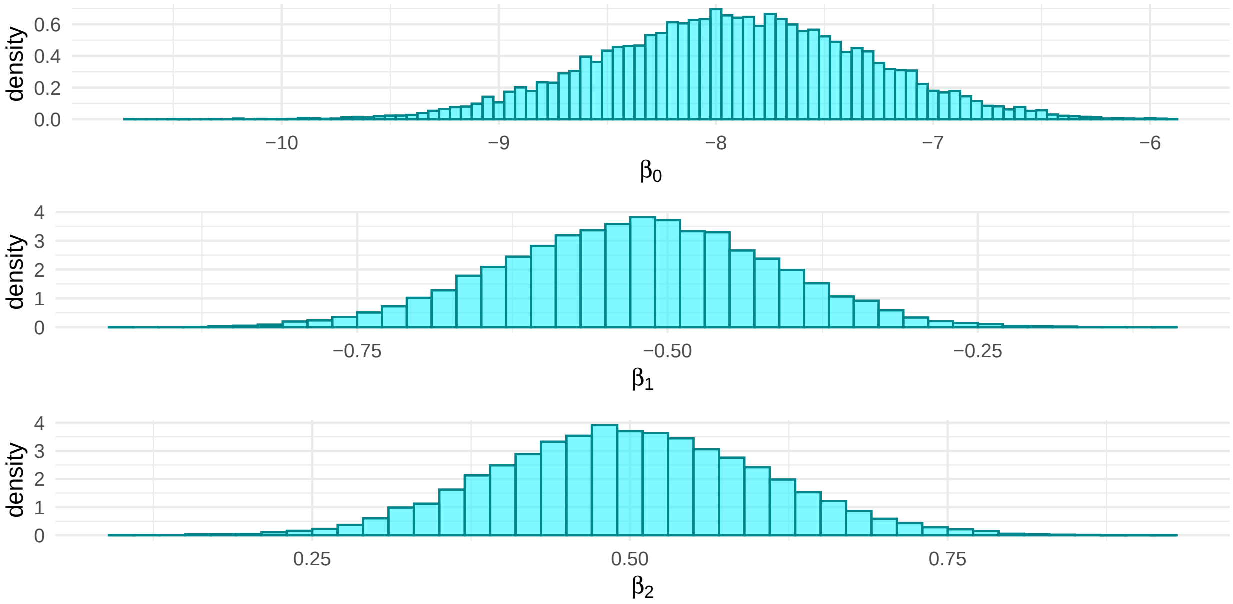
```
rwmh_multivariate_log <- function(start, niter, logtarget, vcov, thinning, burnin){
    # Set current z to the initial point and calculate its log target to save computations
    z  <- start; pz <- logtarget(start)
    # create vector deciding iterations where we record the samples
    store <- seq(from=(1+burnin), 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, so that the algorithm won't waste in regions of the state space that have no mass due to a poor initial guess.

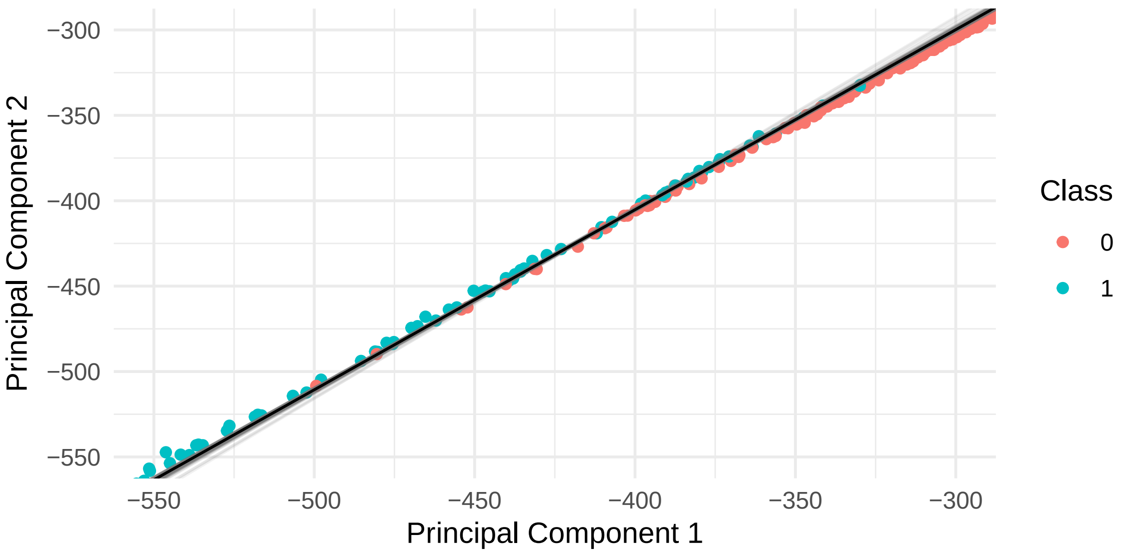
	MLE	MAP
BFGS	0.9473684	0.9473684
NM	0.9473684	0.9385965



The histograms of each coordinate of the samples show RWMH is sampling sensibly.



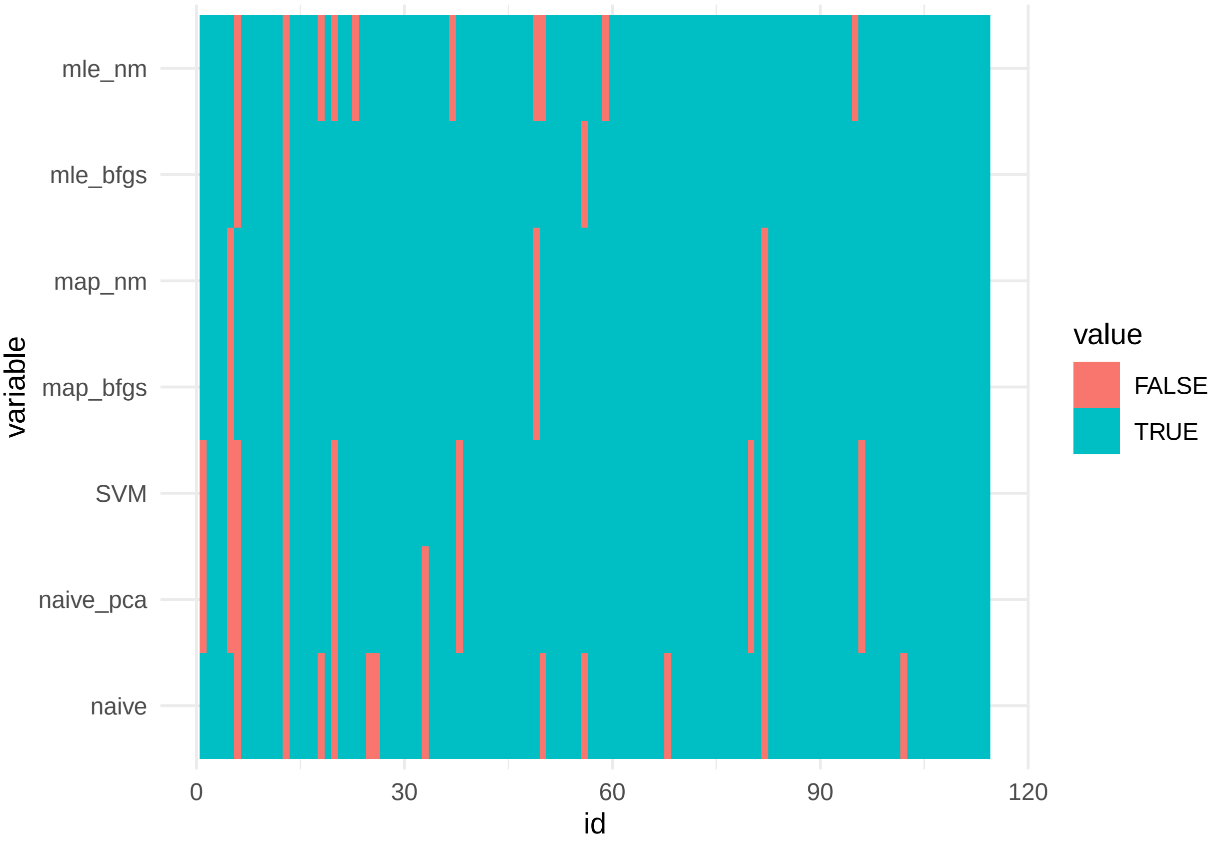
The next thing to do is to plot the decision boundary given by β_{MAP} and the decision boundaries given by (some) of the samples from the posterior.



We can see that Logistic Regression finds a decision boundary that does not reflect what we would expect it to be. In particular, we can see that the points seem to lie on a line with very little variance along the perpendicular direction and since, differently from SVMs, Logistic Regression has no concept of a margin, it settles for a boundary that would easily allow points to be misclassified if they were perturbed along the perpendicular direction. This is also reflected in the confidence of the posterior distribution, having samples which give very similar decision boundaries.

Conclusion

Visualize which are the observations that the models missclassify.


mle_nm
mle_bfgs
map_nm
map_bfgs
SVM
naive_pca
naive
variable
id
value
FALSE
TRUE