KMLMM

Modelling with Relevance Vector Machines

Barcelona, January 16th 2021

Jake Watson

Abstract

The support vector machine (SVM) is a well-known technique for machine learning problems, providing good generalisation with a sparse representation. It also has the disadvantages of being unable to provide probabilistic outputs, and requiring its kernel functions to fulfil stringent conditions. One approach to deal with these issues is the relevance vector machine (RVM), a kernelized Bayesian general linear model, which provides similar generalisation performance and increased sparsity, and includes probabilistic outputs by design.

Introduction

Both of our models fall under the category of *supervised learning*. We are given a set of input vectors $\{\boldsymbol{x}_n,\ t_n\}_{n=1}^{N}$, where x_n is the set of features for an example and t_n is the target values, either a numerical value for regression or a class label for classification.

To make predictions, we wish to *learn* a model of the relationship between the target values and the features, while excluding the random noise present in the features, or *overfitting*. We can formulate a prediction in the following form:

$$y(\boldsymbol{x}; \boldsymbol{w}) = \sum_{x} w \cdot \phi(\boldsymbol{x}) = \mathbf{w}^{\mathrm{T}}\phi(\boldsymbol{x})$$

Where the prediction is a linear combination of a set of basis functions. Our learning objective is then to find values of $\mathbf{w}$ that fit the data.

Support Vector Machines

The SVM attempts to learn a set of weights that minimise the error on the predictions for the training set, while maximising the margin between the two classes in the kernel space. This approach has been very successful, with SVMs being used in hundreds of papers for several decades. However, it still has several limitations:

- *Sparsity*: the number of basis functions, or *support vectors*, in the model representation is still too large. The number of SVs needed increases with the size of the data set, so predictions can be slow.
- *Probabilistic*: to deduce a probability distribution from an SVM model is complex and potentially unreliable [1].
- *Hyperparameter Optimisation*: we must estimate several hyperparameters for SVMs, increasing the computation time required.
- *Kernel Conditions*: any kernel used must satisfy Mercer's condition – it must be continuous, symmetrical and positive definite.

Bayesian General Linear Models

A Bayesian general linear model (GLM) is a method used to find the posterior distribution for the model parameters, given a set of training data, and assuming normally distributed errors. Given the formula:

$$y = \mathcal{N}(\beta^T X, \sigma^2 I)$$

Where y is the prediction value, X is the training set and β is the set of weights for each vector. The value of y is then generated by a Gaussian distribution characterised by a mean and variance. We then aim to find the posterior distribution of β given the set of training data and outputs:

$$P(\beta|y, X) = \frac{P(y|\beta, X) * P(\beta|X)}{P(y|X)}$$

Where $P(y|\beta, X)$ is the likelihood of the data given β and y, multiplied by the prior distribution of β, $P(\beta|X)$. The likelihood of the data set is

$$p(y|\beta, \sigma^2) = \prod_{n=1}^{N} (2\pi\sigma^2)^{-\frac{N}{2}} \cdot \exp\left(\frac{1}{2\sigma^2}\|y - \beta X\|^2\right)$$

This method gives us a posterior distribution for the model parameters, so we can inherently quantify uncertainty in our model. By including a By minimising the negative log-likelihood of this dataset, we can obtain values of the parameters β and σ (maximum-likelihood estimation). Without explicit complexity control, this approach can lead to severe overfitting.

Relevance Vector Machines

Relevance vector machines [2] are a kernelized version of a Bayesian general linear model. By including a kernel, we can extend the range of functions the approach can model to include non-linear relationships, while retaining the speed of computation. Including the kernel in the dataset likelihood:

$$p(t|\mathbf{w},\sigma^2) = \prod_{n=1}^{N}(2\pi\sigma^2)^{-\frac{N}{2}} \cdot \exp\left(\frac{1}{2\sigma^2}\|\mathbf{t}-\boldsymbol{\phi}\mathbf{w}\|^2\right)$$

Where $\boldsymbol{\phi}$ is the representation of the data in the kernel space, $\mathbf{w}$ is the weights matrix, and t is the set of target values. To avoid the overfitting of the Bayesian GLM, we can define a Gaussian prior over the $\mathbf{w}$ weights matrix:

$$p(\mathbf{w}|\boldsymbol{\alpha}) = \prod_{\boldsymbol{i=0}}^{\boldsymbol{N}} \mathcal{N}(w_i|0,\alpha_i^{-1})$$

Where $\boldsymbol{\alpha}$ is a vector of $N+1$ hyperparameters, one per individual. Obtaining the final posterior over the weights is then obtained from Bayes rule:

$$p(\mathbf{w}|\mathbf{t},\boldsymbol{\alpha},\sigma^2) = (2\pi)^{-\frac{N+1}{2}}|\Sigma|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}(\mathbf{w}-\boldsymbol{\mu})^T\Sigma^{-1}(\boldsymbol{w}-\boldsymbol{\mu})\right)$$

Where $\Sigma = (\boldsymbol{\phi}^{\mathrm{T}}\mathbf{B}\boldsymbol{\phi}+\mathbf{A})^{-1}$, $\boldsymbol{\mu} = \boldsymbol{\Sigma}\boldsymbol{\phi}^{\boldsymbol{T}}\mathbf{B}\mathbf{t}$, $\boldsymbol{A} = diag(\alpha_0,\cdots,\alpha_n)$ and $\mathbf{B} = \sigma^{-2}\mathbf{I}_{\mathrm{N}}$. The *marginal likelihood* of the parameters is then given:

$$p(\mathbf{t}|\boldsymbol{\alpha},\sigma^2) = (2\pi)^{-\frac{N+1}{2}}|\mathbf{B}^{-1}+\boldsymbol{\phi}\mathbf{A}^{-1}\boldsymbol{\phi}^{\mathbf{T}}|^{-\frac{1}{2}}\exp\left(-\frac{1}{2}\mathbf{t}^{\mathbf{T}}(\mathbf{B}^{-1}+\boldsymbol{\phi}\mathbf{A}^{-1}\boldsymbol{\phi}^{\mathbf{T}})^{-1}\mathbf{t}\right)$$

By optimising the marginal likelihood over $\boldsymbol{\alpha}$ and σ^2, we find the maximum of $p(\boldsymbol{\alpha},\sigma^2|\mathbf{t})$. We cannot optimise analytically, so we must use an update schedule to estimate $\boldsymbol{\alpha}$ and σ. We can use:

$$\boldsymbol{\alpha_i^{new}} = \frac{\mathbf{1}-\boldsymbol{\alpha_i}\boldsymbol{\Sigma_{ii}}}{\mu_i^2}$$

While estimating, many α_i approach infinity, which causes $p(w_i|\mathbf{t},\boldsymbol{\alpha},\sigma^2)$ to peak at zero – implying that this basis function contributes nothing to the distribution and so can be removed by setting its weight to zero.

Examples of Regression

Machine learning regression algorithms are often tested on the $sinc(x) = \frac{\sin(|x|)}{|x|}$ function. In this project, 100 random examples were drawn from $(-10, 10)$, and used to investigate the performance of the three approaches detailed above. A gaussian noise with standard deviation of 0.1 was added to the data.

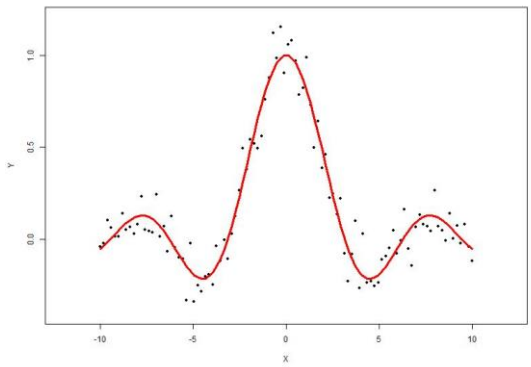

Figure 1: The sinc dataset, with added random noise. The curve is the ideal function.

SVM Regression

The `kernlab` package in R was used to provide a comparison for performance. The code used is below:

```
svmdata <- cbind(X, t)
model <- ksvm(x=X, y=t, kernel="rbfdot")
pred <- predict(model, X)
SVs <- cbind(X[model@SVindex], t[model@SVindex])
```

```
[1] "SVM (radial kernel) regression report:
     mean square error 0.00076262301718958,
     with 69 support vectors"
```

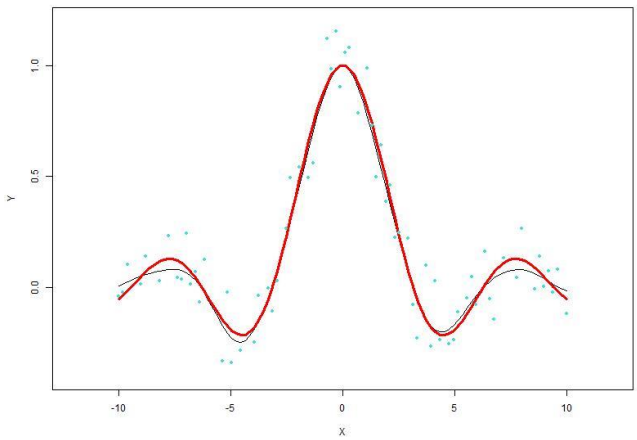

Figure 2: fit to the data for the SVM. The black line is the regression, while the points are the support vectors.

The SVM is clearly able to capture the underlying function well. However, it requires a large number of support vectors to be stored – over two thirds of the original dataset!

GLM

The `arm` package in R was used to provide a comparison for performance against non-kernelized algorithms. The code used is below:

```
non_kernl <- bayesglm(t~ ., data=as.data.frame(svmdata))
pred_nk <- predict(non_kernl)
nk_sinc_err <- MSE(pred_nk, y)
```

```
[1] "NK regression report:
     mean square error 0.00906294455430981"
```

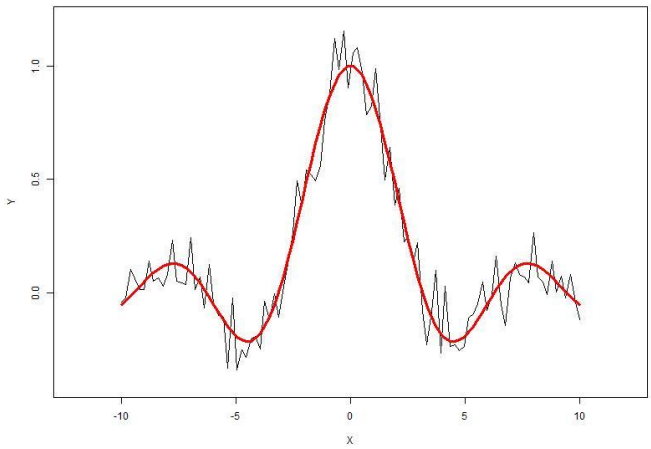

Figure 3: fit to the data using the Bayesian GLM model. Black is the regressed function.

The model is able to capture the complexity of the data, as shown by the very low error. However, it clearly overfits, completely capturing both the underlying variation and the random noise. Clearly the model needs a complexity control to penalise the overfitting.

RVM

An implementation of the relevance vector machine for regression and classification was coded for this project, with the relevant code included as an appendix.

```
results <- RVM(X, t, mode, kernel_type, maxIts)
vectors <- results$vectors
weights <- results$weights
bias <- results$bias

PHI = Kernel(X, X, kernel_type)
y_rvm   = PHI[, vectors] %*% weights + bias;
rvm_sinc_err <- MSE(y_rvm, y)
```

```
[1] "RVM (radial kernel) regression report:
     mean square error 0.00198298791954135,
     with 8 relevance vectors"
```

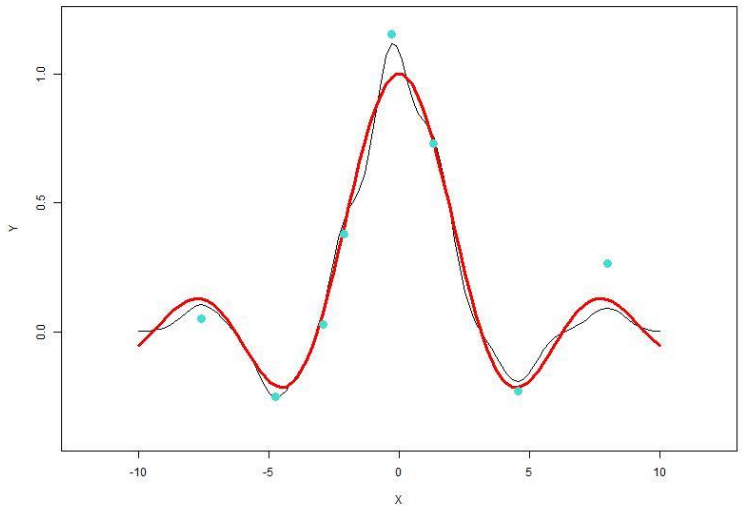

Figure 4: fit to the data using the RVM. The black line is the regressed function, and the points are the relevance vectors.

The RVM has fitted well to the data, with a much lower error than for the GLM, but not as well as for the SVM. Looking at the plot, much of the random noise has also been incorporated. However, the sparsity of the solution is much better than for the SVM, with only 8 relevance vectors, compared to 69 for the SVM.

Classification

To extend the RVM to classification problems, several changes have to be introduced. Firstly, we can use the sigmoid function $sig(x) = 1/(1 + e^{-x})$ and apply it to the prediction to obtain an output in $\{0,1\}$. We must also rewrite the likelihood of the dataset as the Bernoulli distribution:

$$p(\mathbf{t}|\mathbf{w}) = \prod_{n=1}^{N} \sigma\{y(\boldsymbol{x}_n)\}^{t_n}[1 - \sigma\{y(\mathbf{x}_n)\}]^{1-t_n}$$

However, we can't use the same $p(\mathbf{w}|\mathbf{t}, \boldsymbol{\alpha}, \sigma^2)$, as we cannot integrate out the weights. We must use an approximation procedure:

1. For a fixed value of $\boldsymbol{\alpha}$, the most probable weights $\mathbf{w_{MP}}$ are found, giving the mode of the posterior distribution.
2. Given the mode, we can approximate the log-posterior, using Laplace's approximation.

3. Using $\mathbf{w_{MP}}$ and Σ, we can update the hyperparameters $\boldsymbol{\alpha}$ in the same way as for regression.

Once $\boldsymbol{\alpha}$ is updated, we can continue the calculation in the same way as for regression.

Classification Dataset

We can use a standard 2D dataset to test the performance of our algorithms for classification. The Crescent-Moons dataset, with 200 examples, generated using the `RSSL` library in R, is a good non-linear dataset for this task.

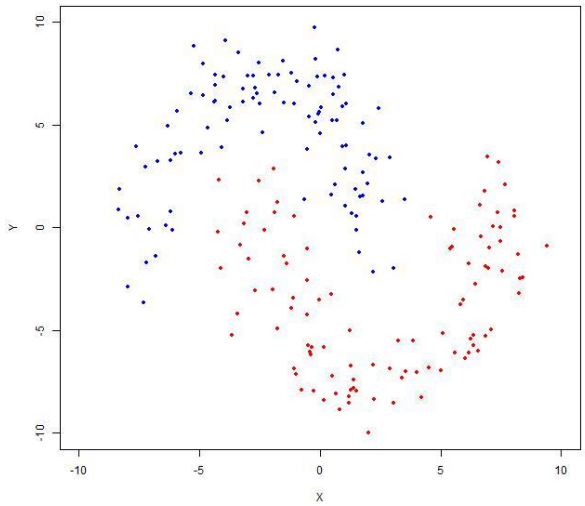

Figure 6: Crescent Moons dataset with 2 classes.

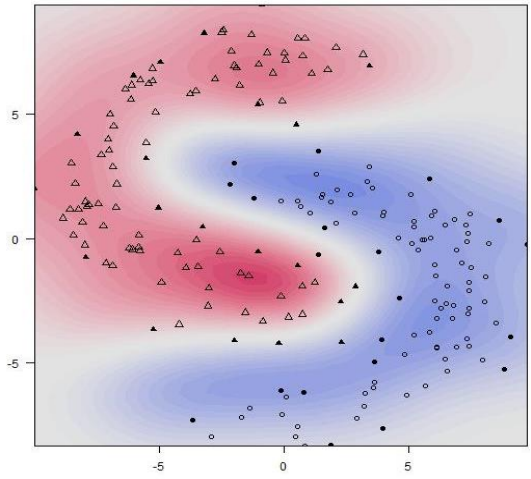

Figure 6: SVM fit to the data for the Crescent Moons dataset.

SVM

```
svmdata <- cbind(X, as.logical(t))
model <- ksvm(x=X, y=t, kernel="rbfdot", type="C-svc")
pred <- predict(model, X)
```

```
[1] "SVM (radial kernel) classification report:
             precision 1,
             recall 1,
             f1-score 0.995024875621891,
             with 41 support vectors"
```

The SVM provides excellent classification performance, with a near-perfect score. Although the plot is mirrored along the diagonal (an irritating issue which could not be removed), the SVM clearly captures the two classes. The support vectors are indicated in the plot by filled circles and triangles. There are fewer support vectors than for the regression task, but we still have a large number.

GLM

```
non_kernl <- bayesglm(t~ ., data=as.data.frame(svmdata))
pred_nk <- predict(non_kernl)
class_nk <- as.matrix(round(pred_nk))
```

```
[1] "NK classification report:
             precision 1,
             recall 1,
             f1-score 1"
```

As before, the GLM fits the data completely, with perfect classification. However, it is clear that it will also overfit the data, given previous performance, so when exposed to new data, the errors will be much higher.

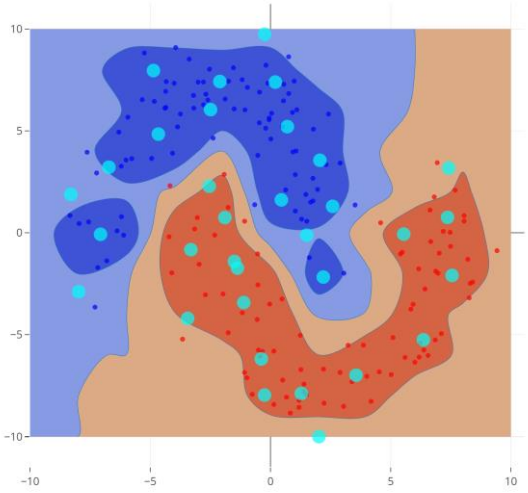

Figure 8: classification by the RVM.

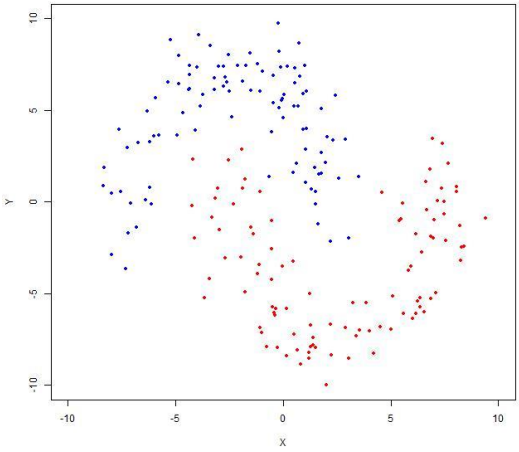

Figure 8: classification by the GLM.

RVM

```
results <- RVM(X, t, mode, kernel_type, maxIts)
vectors <- results$vectors
weights <- results$weights
bias <- results$bias

PHI = Kernel(X, X, kernel_type)
y_moons_rvm = PHI[, vectors] %*% weights + bias;
p_rvm <- sigmoid(y_moons_rvm)
class <- round(p_rvm)
```

```
[1] "RVM (radial kernel) classification report:
             precision 1,
             recall 1,
             f1-score 1,
             with 33 relevance vectors"
```

The RVM obtains a perfect classification score, and as can be seen in the plot, the contours seem reasonable, so we can assume it will generalise fairly well. The number of relevance vectors is only 33, so this is a sparser solution than the SVM.

Discussion

Given the data, we can now discuss the performance of the RVM against the SVM, and the non-kernelized version, the GLM. The RVM performed well at both tasks, with comparable accuracy to the SVM, and a notable increase in sparsity, with fewer relevance vectors in each model. Indeed, it outperforms the SVM in classification, although without application to a test set this is not reliable. It also lacks hyperparameters to set, an advantage over the SVM. With continued tweaks to the code, particularly experimenting with the pruning threshold and the convergence parameter, it may be possible to outperform the SVM. Compared to the GLM, the accuracy and generalisation is clearly superior, with far less overfitting, thanks to the regularisation effect of the Gaussian prior.

However, several shortcomings were noted. Firstly, the training time was much longer for the RVM, particularly in classification. The RVM requires the inversion of an $M \times M$ matrix, with complexity $O(M^3)$, while an SVM training is $O(M^2)$, and the classification mode requires the iterative calculation of the posterior, greatly increasing computational time. This implementation has not been optimized, and a newer version [3] of the RVM was proposed in 2001, with an improved running time, so these concerns may be addressed in future work. It is also noted that the RVM can get stuck in local optimum, a disadvantage compared to the SVM.

Conclusions

The RVM was successfully implemented and compared to its non-kernelized version, the Bayesian GLM, and the SVM, for both classification and regression. It was found to outperform the GLM in both tasks, and the SVM for classification, with comparable results in regression. It also produced sparser solutions than the SVM, and can produce posterior distributions over the weights, which the SVM can only estimate. Given the similar functional form to the SVM, which has been applied in a great variety of work, the RVM presents a promising technique for a broad range of machine learning problems.

1 References

[1] M. Murat, “Can you interpret probabilistically the output of a Support Vector Machine?,” 12 October 2019. [Online]. Available: https://mmuratarat.github.io/2019-10-12/probabilistic-output-of-svm. [Accessed 12 January 2021].

[2] M. E. Tipping, "The Relevance Vector Machine," in *NIPS*, Cambridge, 1999.

[3] M. E. Tipping, "Sparse Bayesian Learning and the Relevance Vector Machine," *Journal of Machine Learning Research,* vol. 1, pp. 211-244, 2001.

Appendix

```
# Estimates the relevance vectors and their weights
#
# OUTPUTS
#   vectors      indices of the relevance vectors in the training set
#   weights      weights of the relevance vectors
#
# INPUTS
#   phi     data representation in kernel basis form
#   t       target values
#   mode    string, either (REGRESSION | CLASSIFICATION)
#   kernel  string, kernel type (distance | gauss | radial)
#   alpha   initial hyperparameters
#   max_it   maximum iterations
#
RVM_train <- function(PHI, t, mode, alpha, kernel, max_it) {

  N <- nrow(PHI)
  M <- ncol(PHI)

  w <- matrix(0L, nrow = M, ncol = 1)
  alpha <- matrix(alpha, nrow = M, ncol = 1)
  gamma <- matrix(1L, nrow = M, ncol = 1)
  PHIt <- t(PHI) %*% t

  # convergence parameter
  CONVERGENCE = 1E-3

  # maximum alpha size
  # if alpha exceeds for a vector,
  # prune that vector
  PRUNING_THRESHOLD = 1e9

  LAST_IT <- FALSE
  REGRESSION <- strcmp(mode, "REGRESSION")

  epsilon <- std(t) * 10/100
  beta <- 1/epsilon^2

  for (i in 1:max_it) {

    # do pruning
    useful_indices <- which(alpha < PRUNING_THRESHOLD)
    useless_indices <- which(alpha >= PRUNING_THRESHOLD)
    useful <- alpha[useful_indices]
    M <- length(useful_indices)
    w[useless_indices] <- 0L
    PHI_used <- PHI[, useful_indices]

    if (REGRESSION){
```

```
      # determine gaussian likelihood
      Hessian <- (t(PHI_used) %*% PHI_used) * beta + diag(useful)
      U <- chol(Hessian)
      Ui <- solve(U)
      w[useful_indices] <- (Ui %*% (t(Ui) %*%
PHIt[useful_indices]))*beta
      ED <- sum(t - (PHI_used %*% w[useful_indices])^2)
      dataLikelihood <- (N*log(beta) - beta*ED)/2
    } else {
      # for classification we use a bernoulli likelihood
      # we call the posterior mode finder
      post <- posterior(w[useful_indices], alpha[useful_indices],
PHI_used, t, 75)
      w[useful_indices] <- post$weight
      Ui <- post$Ui
      dataLikelihood <- post$lMode
    }

    diagSig <- rowSums(Ui^2)
    gamma <- 1 - useful*diagSig

    if (!LAST_IT) {

      # MacKay updates of hyperparameters
      oldAlpha <- log(alpha[useful_indices])
      alpha[useful_indices] <- gamma / (w[useful_indices]^2)
      au <- alpha[useful_indices]

      # check for convergence of log hyperparameters
      maxDAlpha <- max(abs(oldAlpha[which(au != 0)]) - log(au[which(au
!= 0)]))

      if (maxDAlpha < CONVERGENCE) {
        print(paste("Reached convergence in ", i, "iterations"))
        LAST_IT <- TRUE
      }

      if (REGRESSION) {
        beta = (N - sum(gamma)/ED)
      }

    } else {
      print("Max iterations reached")
      break
    }
  }

  weights <- w[useful_indices]
  vectors <- useful_indices

  return(list("vectors" = vectors, "weights" = weights))
}

# Finds the mode of the posterior distribution
#
# OUTPUTS
#   w       weights for each vector, maximises the posterior
#   Ui      inverse of the cholesky factor of the hessian
#   lMode   log likelihood of the data at the mode
#
# INPUTS
```

```
#   phi     data representation in kernel basis form
#   t       target values
#   w       initial vector weights
#   alpha   initial hyperparameters
#   iters   maximum iterations
#
posterior <- function(w, alpha, phi, t, iters) {

  # convergence parameter
  GRAD_STOP <- 1e-6
  # minimum search step
  LAMBDA_MIN <- 2^(-8)

  N <- nrow(phi)
  d <- ncol(phi)
  M <- length(w)

  A <- diag(alpha)
  errs <- matrix(0L, nrow = iters, ncol = 1)
  PHIw <- phi %*% w
  y = sigmoid(as.matrix(PHIw))

  # initial log posterior value
  data_term <- -(sum(log(y[which(t == 1)])) + sum(log(1-
y[which(t==0)])))/N
  regulariser <- (t(alpha) %*% (w^2))/(2*N)
  err_new <- data_term + regulariser

  for (i in 1:iters) {
    yvar <- y * (1 - y)
    PHIV <- phi * (yvar %*% matrix(1L, nrow = 1, ncol = d))
    e <- (t - y)

    # initial gradient vector and hessian
    g <- (t(phi) %*% e) - (alpha * w)
    Hessian <- (t(PHIV) %*% phi) + A

    if (i==1) {
      if (rcond(Hessian) < (2^-52)){
        stop("ill conditioned Hessian")
      }
    }

    errs[i] <- err_new

    # check for convergence
    if (i > 2 & norm(g)/M < GRAD_STOP) {
      errs <- errs[1:i]
      print(paste("posterior converged after ", i, " iterations"))
      break
    }

    # Newton step

    U <- chol(Hessian)
    delta_w <- mldivide(U, mldivide(t(U), g))
    lambda = 1

    while (lambda > LAMBDA_MIN) {
      w_new <- w + lambda*delta_w
      PHIw <- phi %*% w_new
```

```
      y <- sigmoid(PHIw)

      # compute new error
      if (any(y[which(t == 1)] == 0) | any(y[which(t == 0)] == 1)){
        err_new = Inf
      } else {
        data_term <- -(sum(log(y[which(t == 1)])) + sum(log(1 -
y[which(t == 0)])))/N
        regulariser <- (t(alpha) %*% (w_new^2))/(2*N)
        err_new <- data_term + regulariser
      }

      # if the error has increased, reduce the step size
      if (err_new > errs[i]) {
        lambda = lambda/2
      } else {
      # error decreased, accept the step
        w = w_new
        lambda = 0
      }
    }

    # we have converged close to the optimum and cannot take any more
steps
    if (lambda) {
      break
    }
  }

  # calculate final values and return
  Ui <- solve(U)
  lMode <- -N*data_term

  return(list("weight" = w, "Ui" = Ui, "lMode" = lMode))
}
```