Kernel Methods

Max Turgeon

STAT 4690–Applied Multivariate Analysis

Motivation

- Linearity has been an important assumption for most of the multivariate methods we have discussed.
    - Multivariate Linear Regression
    - PCA, FA, CCA
- This assumption may be more realistic after a transformation of the data.
    - E.g. Log transformation
    - Embedding in a higher dimensional space?

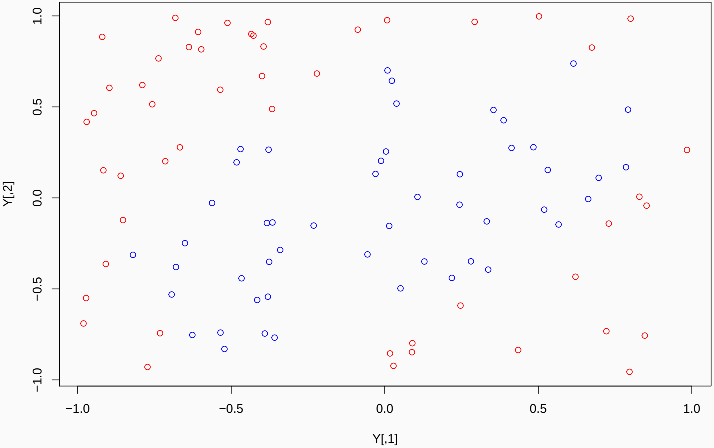
Example i

```
set.seed(1234)
n <- 100
# Generate uniform data
Y <- cbind(runif(n, -1, 1),
           runif(n, -1, 1))

# Check if it falls inside ellipse
Sigma <- matrix(c(1, 0.5, 0.5, 1), ncol = 2)
dists <- sqrt(diag(Y %*% solve(Sigma) %*%
                     t(Y)))
inside <- dists < 0.85
```

Example ii

```r
# Plot points
colours <- c("red", "blue")[inside + 1]
plot(Y, col = colours)
```

Example iii

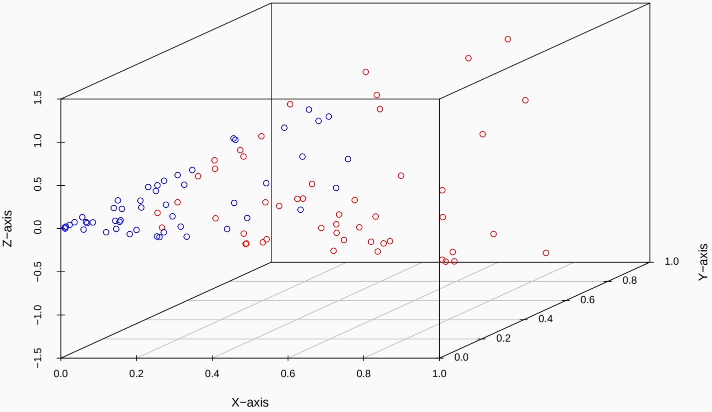


Example iv

```r
# Transform data
# (X, Y) -> (X^2, Y^2, sqrt(2)*X*Y)
Y_transf <- cbind(Y[,1]^2, Y[,2]^2,
                  sqrt(2)*Y[,1]*Y[,2])
```

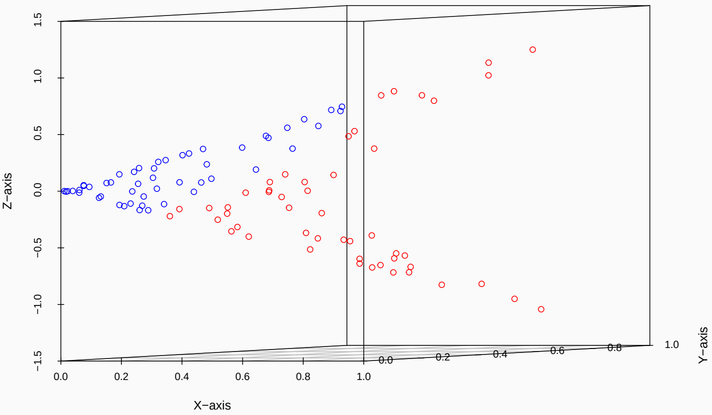
```r
library(scatterplot3d)
scatterplot3d(Y_transf, color = colours,
              xlab = "X-axis",
              ylab = "Y-axis",
              zlab = "Z-axis")
```

Example v



Z−axis
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
X−axis
0.0
0.2
0.4
0.6
0.8
1.0
Y−axis
0.0
0.2
0.4
0.6
0.8
1.0

Example vi



Example vii

```r
# Linear regression
outcome <- ifelse(inside, 1, -1)
head(outcome)
```

```
## [1] -1  1  1 -1 -1  1
```

```r
model1 <- lm(outcome ~ Y)
pred1 <- sign(predict(model1))
table(outcome, pred1) # 67%
```

Example viii

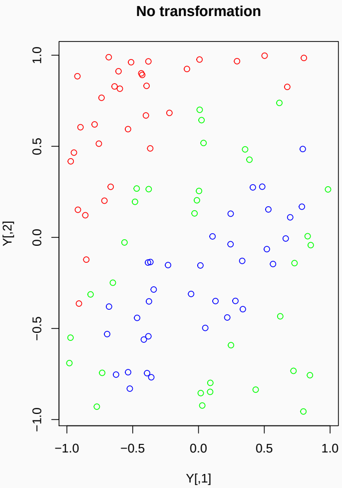
```
##         pred1
## outcome -1  1
##      -1 32 18
##      1  15 35
```

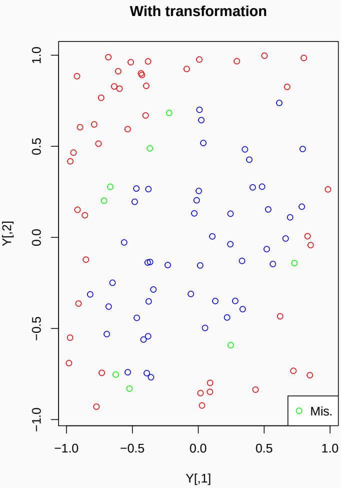
```
model2 <- lm(outcome ~ Y_transf)
pred2 <- sign(predict(model2))
table(outcome, pred2) # 92%
```

Example ix

```
##         pred2
## outcome -1  1
##      -1 44  6
##      1   2 48
```

Example x


No transformation
Y[,2]
Y[,1]
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


With transformation
Y[,2]
Y[,1]
Mis.
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0

Overfitting i

- **Overfitting** is "the production of an analysis that corresponds too closely or exactly to a particular set of data, and may therefore fail to fit additional data or predict future observations reliably" (OED)
  - In other words, a model is overfitted if it explains the *training* data very well, but does poorly on *test* data.
- In regression, this often happens when we have too many covariates
  - Too many *parameters* for the sample size

Overfitting ii

- When embedding our covariates into a higher dimensional space, we are increasing the number of parameters.
    - There is a danger of overfitting.
- **One possible solution**: Regularised (or penalised) regression.
    - We constrain the parameter space using a penalty function.

Ridge regression i

- Let $(Y_i, \mathbf{X}_i), i = 1, \ldots, n$ be a sample of outcome with covariates.
- **Univariate Linear Regression**: Assume that we are interested in the linear model

$$Y_i = \beta^T \mathbf{X}_i + \epsilon_i.$$

- The Least-Squares estimate of β is given by

$$\hat{\beta}_{LS} = (\mathbb{X}^T \mathbb{X})^{-1} \mathbb{X} \mathbf{Y},$$

where

Ridge regression ii

$$\mathbb{X}^T = \begin{pmatrix} \mathbf{X}_1 & \cdots & \mathbf{X}_n \end{pmatrix},$$
$$\mathbf{Y} = (Y_1, \ldots, Y_n).$$

- If the matrix $\mathbb{X}^T\mathbb{X}$ is almost singular, then the least-squares estimate will be unstable.
- **Solution**: Add a small quantity along its diagonal.
  - $\mathbb{X}^T\mathbb{X} \to \mathbb{X}^T\mathbb{X} + \lambda I$
  - Bias-Variance trade-off

Ridge regression iii

- The Ridge estimate of β is given by

$$\hat{\beta}_R = (\mathbb{X}^T\mathbb{X} + \lambda I)^{-1}\mathbb{X}^T\mathbf{Y}.$$

Example i

```r
library(ElemStatLearn)
library(tidyverse)

data_train <- prostate %>%
  filter(train == TRUE) %>%
  dplyr::select(-train)
data_test <- prostate %>%
  filter(train == FALSE) %>%
  dplyr::select(-train)
```

Example ii

```r
model1 <- lm(lpsa ~ .,
             data = data_train)
pred1 <- predict(model1, data_test)

mean((data_test$lpsa - pred1)^2)
```

```
## [1] 0.521274
```

Example iii

```r
# glmnet does lasso, elastic-net and
# ridge regression
library(glmnet)
X_train <- model.matrix(lpsa ~ . - 1,
             data = data_train)
model2 <- glmnet(X_train, data_train$lpsa,
                 alpha = 0, lambda = 0.7)
```

Example iv

```
X_test <- model.matrix(lpsa ~ . -  1,
              data = data_test)
pred2 <- predict(model2, X_test)

mean((data_test$lpsa - pred2)^2)
```

```
## [1] 0.5060843
```

Dual problem i

- The Ridge estimate actually minimises a **regularized** least-squares function:

$$RLS(\beta) = \frac{1}{2}\sum_{i=1}^{n}(Y_i - \beta^T \mathbf{X}_i)^2 + \frac{\lambda}{2}\beta^T\beta.$$

- If we take the derivative with respect to β, we get

$$\frac{\partial}{\partial\beta}RLS(\beta) = -\sum_{i=1}^{n}(Y_i - \beta^T \mathbf{X}_i)\mathbf{X}_i + \lambda\beta.$$

- Setting it equal to 0 and rearranging, we get

$$\beta = \frac{1}{\lambda}\sum_{i=1}^{n}(Y_i - \beta^T \mathbf{X}_i)\mathbf{X}_i.$$

Dual problem ii

- Define $a_i = \frac{1}{\lambda}(Y_i - \beta^T \mathbf{X}_i)$. We then get

$$\beta = \sum_{i=1}^{n} a_i \mathbf{X}_i = \mathbb{X}^T \alpha,$$

  where $\alpha = (a_1, \ldots, a_n)$.
- **Why?** We can now rewrite $RLS(\beta)$ as a function of α. First note that

$$RLS(\beta) = \frac{1}{2}(\mathbf{Y} - \mathbb{X}\beta)^T(\mathbf{Y} - \mathbb{X}\beta) + \frac{\lambda}{2}\beta^T\beta.$$

Dual problem iii

- Now we can substitute $\beta = \mathbb{X}^T\alpha$:

$$\begin{aligned} RLS(\alpha) &= \frac{1}{2}(\mathbf{Y} - \mathbb{X}\mathbb{X}^T\alpha)^T(\mathbf{Y} - \mathbb{X}\mathbb{X}^T\alpha) + \frac{\lambda}{2}(\mathbb{X}^T\alpha)^T(\mathbb{X}^T\alpha) \\ &= \frac{1}{2}(\mathbf{Y} - (\mathbb{X}\mathbb{X}^T)\alpha)^T(\mathbf{Y} - (\mathbb{X}\mathbb{X}^T)\alpha) + \frac{\lambda}{2}\alpha^T(\mathbb{X}\mathbb{X}^T)\alpha. \end{aligned}$$

- This formulation of regularised least squares in terms of α is called the **dual problem**.
- **Key observation**: $RLS(\alpha)$ depends on X_i *only* through the Gram matrix $\mathbb{X}\mathbb{X}^T$.
    - If we all we know are the dot products of the covariates X_i, we can still solve the ridge regression problem.

Kernel ridge regression i

- Suppose we have a transformation $\Phi : \mathbb{R}^p \to \mathbb{R}^N$, where N is typically larger than p and can even be infinity.
- Let K be the $n \times n$ matrix whose (i, j)-th entry is the dot product between $\Phi(\mathbf{X}_i)$ and $\Phi(\mathbf{X}_j)$:

$$K_{ij} = \Phi(\mathbf{X}_i)^T \Phi(\mathbf{X}_j).$$

- **Important observation**: This actually induces a map on pairs of points in $\mathbb{R}^p$:

$$k(\mathbf{X}_i, \mathbf{X}_j) = \Phi(\mathbf{X}_i)^T \Phi(\mathbf{X}_j).$$

- We will call the function k the **kernel function**.

Kernel ridge regression ii

- Now, we can use the dual formulation of ridge regression to fit a linear model between Y_i and the transformed $\Phi(X_i)$:

$$Y_i = \beta^T \Phi(\mathbf{X}_i) + \epsilon_i.$$

- By setting the derivative of $RLS(\alpha)$ equal to zero and solving for α, we see that

$$\hat{\alpha} = (K + \lambda I_n)^{-1}\mathbf{Y}.$$

Kernel ridge regression iii

- Note that we would need to know all the images $\Phi(\mathbf{X}_i)$ to recover $\hat{\beta}$ from $\hat{\alpha}$. On the other hand, we don't actually need $\hat{\beta}$ to obtain the *fitted* values:

$$\hat{\mathbf{Y}} = \Phi(\mathbb{X})\hat{\beta} = \Phi(\mathbb{X})\Phi(\mathbb{X})^T\hat{\alpha} = K\hat{\alpha}.$$

- To obtain the predicted value for a new covariate profile $\tilde{\mathbf{X}}$, first compute all the dot products in the feature space:

$$\mathbf{k} = (k(\mathbf{X}_1, \tilde{\mathbf{X}}), \ldots, k(\mathbf{X}_n, \tilde{\mathbf{X}})).$$

Kernel ridge regression iv

- We can then obtain the predicted value:

$$\begin{aligned}\tilde{Y} &= \hat{\beta}^T \Phi(\tilde{\mathbf{X}}) \\ &= \hat{\alpha}^T \Phi(\mathbb{X})\Phi(\tilde{\mathbf{X}}) \\ &= \hat{\alpha}^T \mathbf{k} \\ &= \mathbf{k}^T (K + \lambda I_n)^{-1}\mathbf{Y}.\end{aligned}$$

Example (cont'd) i

```r
# Let's start with the identity map for Phi
# We should get the same results as Ridge regression
X_train <- model.matrix(lpsa ~ .,
                        data = data_train)
Y_train <- data_train$lpsa
```

Example (cont'd) ii

```
# Ridge regression
beta_hat <- solve(crossprod(X_train) +
                    0.7*diag(ncol(X_train))) %*%
  t(X_train) %*% Y_train

beta_hat[1:3]
```

```
## [1] 0.1323063 0.5709660 0.6160020
```

Example (cont'd) iii

```r
# Dual problem
alpha_hat <- solve(tcrossprod(X_train) +
                     0.7*diag(nrow(X_train))) %*%
  Y_train

(t(X_train) %*% alpha_hat)[1:3]
```

```
## [1] 0.1323063 0.5709660 0.6160020
```

```r
all.equal(beta_hat, t(X_train) %*% alpha_hat)
```

```
## [1] TRUE
```

Important observation

- We assumed that we had an embedding of the data into a higher dimensional space.
- But our derivation only required the **dot products** of our observations in the feature space.
- Therefore, we don't need to explicitly define the transformation.
- All we need is to define a **kernel function**.

Definition

- We need a way to test whether a function $k(\mathbf{X}_i, \mathbf{X}_j)$ is a valid kernel, i.e. that it arises from a dot product in some higher dimensional space.
- **Theorem**: A necessary and sufficient condition for $k(\cdot, \cdot)$ to be a valid kernel is that the Gram matrix K, whose (i, j)-th element is $k(\mathbf{X}_i, \mathbf{X}_j)$, is positive semidefinite for all choices of subsets $\{\mathbf{X}_1, \ldots, \mathbf{X}_n\}$ from the sample space.
    - In particular, $k(\cdot, \cdot)$ needs to be *symmetric*.

Examples of valid kernels

1. **Polynomial kernel**: $k(\mathbf{X}_i, \mathbf{X}_j) = (\mathbf{X}_i^T \mathbf{X}_j + c)^d$, for a non-negative real number c and d is a positive integer.
2. **Sigmoidal kernel**: $k(\mathbf{X}_i, \mathbf{X}_j) = \tanh(a\mathbf{X}_i^T \mathbf{X}_j - b)$, for $a, b > 0$ real numbers.
3. **Gaussian kernel**: $k(\mathbf{X}_i, \mathbf{X}_j) = \exp\left(-\|\mathbf{X}_i - \mathbf{X}_j\|^2/2\sigma^2\right)$, where $\sigma^2 > 0$.

In general, kernel functions measure **similarity** between the inputs.

And note that the inputs need not be elements of $\mathbb{R}^p$: you can define kernel functions on strings (for NLP) and graphs (for network analysis).

Combining kernels i

A powerful of creating new kernels is by combining old ones: let k_1, k_2 be kernels, c a constant, A a positive semidefinite matrix, and f a real-valued function. Then the following are also valid kernels:

1. $k(\mathbf{X}_i, \mathbf{X}_j) = ck_1(\mathbf{X}_i, \mathbf{X}_j)$
2. $k(\mathbf{X}_i, \mathbf{X}_j) = f(\mathbf{X}_i)k_1(\mathbf{X}_i, \mathbf{X}_j)f(\mathbf{X}_i)$
3. $k(\mathbf{X}_i, \mathbf{X}_j) = \exp(k_1(\mathbf{X}_i, \mathbf{X}_j))$
4. $k(\mathbf{X}_i, \mathbf{X}_j) = k_1(\mathbf{X}_i, \mathbf{X}_j) + k_2(\mathbf{X}_i, \mathbf{X}_j)$
5. $k(\mathbf{X}_i, \mathbf{X}_j) = k_1(\mathbf{X}_i, \mathbf{X}_j)k_2(\mathbf{X}_i, \mathbf{X}_j)$
6. $k(\mathbf{X}_i, \mathbf{X}_j) = \mathbf{X}_i A \mathbf{X}_j$

Choosing a kernel function

- With so many choices, how can we choose the *right* kernel?
- One approach is to use a kernel that measures similarity in a manner relevant to your problem:
    - For polynomial kernels with $c = 0$, they are invariant to orthogonal transformations of the feature space.
- There are also ways of combining the results from different kernels:
    - **Prediction**: Ensemble methods
    - **Inference**: Omnibus tests

Example (cont'd) i

```
library(kernlab)
```

```
##
## Attaching package: 'kernlab'

## The following object is masked from 'package:purrr'
##
##     cross
```

Example (cont'd) ii

```
## The following object is masked from 'package:ggplot2
##
##     alpha
```

```
# Let's use the quadratic kernel
poly <- polydot(degree = 2)

Kmat <- kernelMatrix(poly, X_train)
Kmat[1:3, 1:3]
```

Example (cont'd) iii

```
##            1        2        3
## 1  6501734  8712023 14104480
## 2  8712023 11681713 18916284
## 3 14104480 18916284 35263969
```

```r
alpha_poly <- solve(Kmat + 0.7*diag(nrow(X_train))) %*
  Y_train
```

Example (cont'd) iv

```r
# Let's predict the test data
X_test <- model.matrix(lpsa ~ .,
                       data = data_test)
k_pred <- kernelMatrix(poly, X_train, X_test)

pred_poly <- drop(t(alpha_poly) %*% k_pred)
mean((data_test$lpsa - pred_poly)^2)
```

```
## [1] 1.007974
```

Example (cont'd) v

```r
# Compare with linear kernel
pred_lin <- drop(t(alpha_hat) %*%
                   tcrossprod(X_train, X_test))
mean((data_test$lpsa - pred_lin)^2)
```

```
## [1] 0.5180924
```

Example (cont'd) vi

```
# Now let's try a Gaussian kernel
# Note: Look at documentation for
# parametrisation
rbf <- rbfdot(sigma = 0.05)
Kmat <- kernelMatrix(rbf, X_train)

alpha_rbf <- solve(Kmat + 0.7*diag(nrow(X_train))) %*%
  Y_train
```

Example (cont'd) vii

```
k_pred <- kernelMatrix(rbf, X_train, X_test)

pred_rbf <- drop(t(alpha_rbf) %*% k_pred)
mean((data_test$lpsa - pred_rbf)^2)
```

```
## [1] 3.530104
```

Example (cont'd) viii

```
# Can we do better by choosing a different sigma?
n <- nrow(X_train)

fit_rbf <- function(sigma) {
  rbf <- rbfdot(sigma = sigma)
  Kmat <- kernelMatrix(rbf, X_train)
  alpha_rbf <- solve(Kmat + 0.7*diag(n)) %*%
    Y_train
  return(list(alpha = alpha_rbf,
              rbf = rbf))
}
```

Example (cont'd) ix

```r
pred_rbf <- function(fit) {
  k_pred <- kernelMatrix(fit$rbf, X_train,
                         X_test)
  pred_rbf <- drop(t(fit$alpha) %*% k_pred)
  return(pred_rbf)
}
```

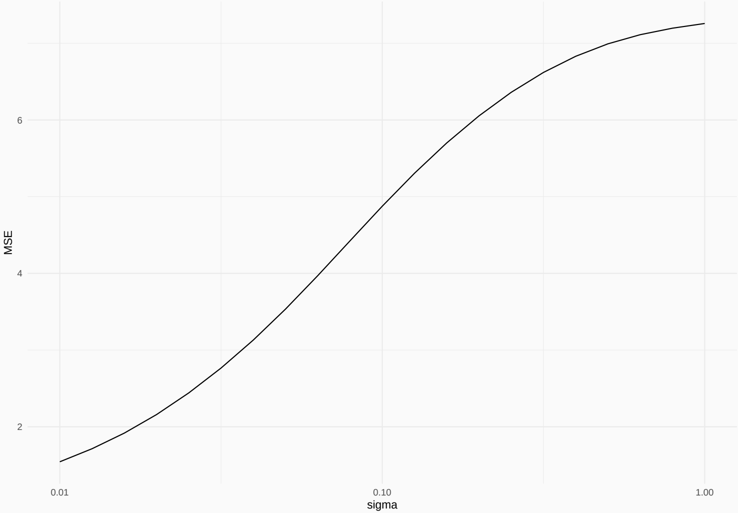
Example (cont'd) x

```r
sigma_vect <- 10^seq(0, -2, by = -0.1)
MSE <- sapply(sigma_vect,
              function (sigma) {
                fit_rbf <- fit_rbf(sigma)
                pred_rbf <- pred_rbf(fit_rbf)
                mean((data_test$lpsa - pred_rbf)^2)
                })
```

Example (cont'd) xi

```
data.frame(
  sigma = sigma_vect,
  MSE = MSE
) %>%
  ggplot(aes(sigma, MSE)) +
  geom_line() +
  theme_minimal() +
  scale_x_log10()
```

Example (cont'd) xii



Example (cont'd) xiii

```r
data.frame(
  sigma = sigma_vect,
  MSE = MSE
) %>%
  filter(MSE == min(MSE))
```

```
##   sigma      MSE
## 1  0.01 1.543654
```

Cross-validation i

- In the example above, we saw that we can tune (or select) the parameter `sigma` by fitting various models and computing the resulting MSE on the **test** data.
- Note that this is the most accurate way to estimate the generalization capabilities of your model.
- In practice, if you don't have enough data to set aside a testing dataset, you can use **cross-validation** to derive a similar estimate.

Cross-validation ii

Algorithm
Let $K > 1$ be a positive integer.

1. Separate your data into K subsets of (approximately) equal size.
2. For $k = 1, \ldots, K$, put aside the k-th subset and use the remaining $K - 1$ subsets to train your algorithm.
3. Using the trained algorithm, predict the values for the held out data.
4. Calculate MSE_k as the Mean Squared-Error for these predictions.
5. The overall MSE estimate is given by

$$MSE = \frac{1}{K}\sum_{k=1}^{K} MSE_k.$$

Example i

```r
library(caret)
```

```
## Loading required package: lattice

##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr'
##
##     lift
```

Example ii

```r
# Blood-Brain barrier data
data(BloodBrain)
length(logBBB)
```

```
## [1] 208
```

```r
dim(bbbDescr)
```

```
## [1] 208 134
```

Example iii

```r
# 5-fold CV with sigma = 0.05
trainIndex <- createFolds(logBBB, k = 5)
str(trainIndex)
```

```
## List of 5
##  $ Fold1: int [1:41] 27 30 32 39 46 51 55 70 72 77
##  $ Fold2: int [1:42] 3 7 8 9 15 21 24 25 26 31 ...
##  $ Fold3: int [1:42] 5 6 10 11 13 16 28 29 35 53 ..
##  $ Fold4: int [1:42] 1 2 17 20 22 23 33 34 42 44 ..
##  $ Fold5: int [1:41] 4 12 14 18 19 36 37 40 43 47 .
```

Example iv

```r
# Let's redefine our functions from earlier
fit_rbf <- function(sigma, data_train, Y_train) {
  rbf <- rbfdot(sigma = sigma)
  Kmat <- kernelMatrix(rbf, data_train)
  alpha_rbf <- solve(Kmat +
                          0.7*diag(nrow(data_train))) %*%
    Y_train
  return(list(alpha = alpha_rbf, rbf = rbf))
}
```

Example v

```
pred_rbf <- function(fit, data_train, data_test) {
  k_pred <- kernelMatrix(fit$rbf, data_train,
                         data_test)
  pred_rbf <- drop(t(fit$alpha) %*% k_pred)
  return(pred_rbf)
}
```

Example vi

```
sapply(trainIndex, function(index){
         data_train <- bbbDescr[-index,] %>%
           model.matrix( ~ ., data = .)
         Y_train <- logBBB[-index]
         data_test <- bbbDescr[index,] %>%
           model.matrix( ~ ., data = .)
         fit_rbf <- fit_rbf(0.05, data_train, Y_train)
         pred_rbf <- pred_rbf(fit_rbf, data_train,
                                     data_test)
         mean((logBBB[index] - pred_rbf)^2)
       }) -> MSEs
```

Example vii

```
MSEs
```

```
##     Fold1     Fold2     Fold3     Fold4     Fold5
## 0.7705720 0.7075762 0.4702274 0.5828595 0.4590561
```

```
mean(MSEs)
```

```
## [1] 0.5980582
```

Example viii

```r
# Now, we can repeat for multiple sigmas
mse_calc <- function(sigma, data_train, data_test,
                     Y_train, Y_test) {
  fit_rbf <- fit_rbf(sigma, data_train, Y_train)
  pred_rbf <- pred_rbf(fit_rbf, data_train,
                       data_test)
  mean((Y_test - pred_rbf)^2)
}
```

Example ix

```
sapply(trainIndex, function(index){
        data_train <- bbbDescr[-index,] %>%
          model.matrix( ~ ., data = .)
        Y_train <- logBBB[-index]
        data_test <- bbbDescr[index,] %>%
          model.matrix( ~ ., data = .)
        sapply(sigma_vect, mse_calc,
               data_train = data_train,
               data_test = data_test,
               Y_train = Y_train,
               Y_test = logBBB[index])}) -> MSEs
```

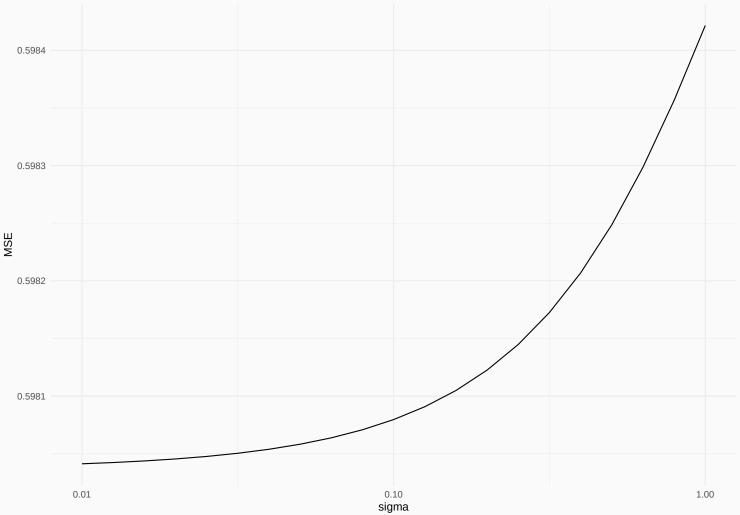
Example x

```r
head(rowMeans(MSEs), n = 4)
```

```
## [1] 0.5984216 0.5983566 0.5982986 0.5982486
```

```r
data.frame(
  sigma = sigma_vect,
  MSE = rowMeans(MSEs)
) %>%
  ggplot(aes(sigma, MSE)) +
  geom_line() +
  theme_minimal() +
  scale_x_log10()
```

Example xi



Example xii

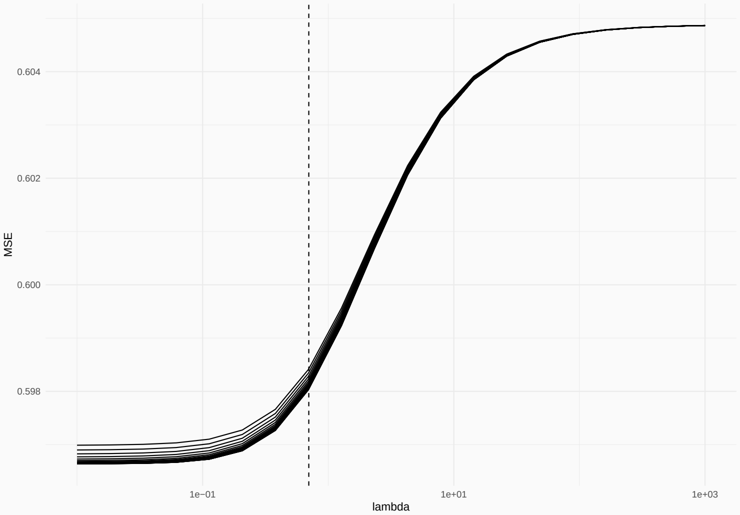
```r
# We can also tune lambda (see R code)
MSEs <- MSEs %>%
  gather(sigma, MSE, -lambda) %>%
  mutate(sigma = as.numeric(sigma))

head(MSEs, n = 3)
```

```
##      lambda sigma       MSE
## 1 1000.0000     1 0.6048670
## 2  545.5595     1 0.6048542
## 3  297.6351     1 0.6048309
```

Example xiii

```r
MSEs %>%
  ggplot(aes(lambda, MSE, group = sigma)) +
  geom_line() +
  theme_minimal() +
  scale_x_log10() +
  geom_vline(xintercept = 0.7, linetype = 'dashed')
```

Example xiv



Example xv

```
MSEs %>%
  filter(MSE == min(MSE))
```

```
##   lambda sigma       MSE
## 1   0.01  0.01 0.5966406
```

Comments

- We can see that the MSE flattens out for all curves around $\lambda \approx 0.1$.
    - Only incremental gains when reducing λ further.
- Similarly, all curves converge to one another for different `sigma`
    - Only incremental gain when reducing `sigma` further.
- For these reasons, we could select $\lambda = 0.01$ and `sigma = 0.01` as our prediction model.
- Note that this gives us better performance than a simple linear model (for which MSE $=$ 1.75).

General comments i

- Finding a good kernel function is difficult, and it involves a lot of trial and error.
  - *One possible strategy*: fit multiple kernels, tune them all, and pick best.
  - *Even better strategy*: fit multiple kernels, tune them all, and **combine the predictions**.
- Unlike traditional methods, kernel methods suffer from *too much data*.
  - Recall that the Gram matrix K is $n \times n$, and so it can become very large.

General comments ii

    - The limitations are mostly computational and related to memory management, and accordingly there are multiple tricks to make it work with “big data”.
- Kernel methods tend to overfit, and therefore it is good practice to *regularise* them using a penalty term (e.g. ridge penalty).
- It’s also good practice to compare kernel methods to simpler methods (e.g. linear regression).
    - If you can’t beat a simple method, what’s the point of a complicated one?