Statistical Learning in Practice (Lent 2020)

Practicals

Dr Alberto J Coca*

March 2, 2020

The computers in GL.04 (CATAM room) are dual boot Ubuntu (Linux) and Windows machines; the ones towards the front use Ubuntu by default, the ones at the back use Windows. If you wish to change operating system, simply reset the computer and choose your preferred option.

The computers are programmed to switch themselves off at night (an energy saving measure). At the start of your session, if your machine is off, then please switch it on. At the end of your session please log out. **Please do not shut down your computer**. This and subsequent worksheets will largely assume that you are using Linux, but there should be little difficulty translating things into Windows or macOS (except for Practical 8: you should use Linux if you get errors).

We will use `R` in all our practicals. It is recommended that you also use `RStudio`, an integrated development environment (IDE) for `R`, and save all your work for each practical in a separate `.r` script. Both `R` and `RStudio` have been pre-installed on all machines in the CATAM room. You can download them for your own computer from the following links.

```
R:          http://cran.r-project.org/
RStudio:    https://www.rstudio.com/
```

Please note that the lectures and practicals may get slightly out of synchronisation, and that you may occasionally do things in the practical that you meet in a later lecture. Use the `R` help system, and ask questions!

*These notes are based on those of Dr Tengyao Wang's from 2018–2019; I thank him for sharing the original version with me. Please, email me if you find any typos/errors.

Contents

Practical 1: Linear regression and Poisson regression

Housing price data

Our first example is a dataset about the Boston housing market. The original full dataset is available from the `mlbench` package. Download the dataset and store it in an `R` dataframe using the following commands (NB: you should not include the symbol `>` from the beginning of each line).

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "housing.csv"
> housing <- read.csv(paste0(filePath, fileName))  # read in the csv file
> head(housing)  # check that the dataframe has been successfully created
```

Try `?read.csv` and `?paste0` for more details of the commands used. A dataframe in `R` is a collection of lists of equal lengths, arranged in a two-dimensional array. The `housing` dataset contains the following variables measured at different census tracts of Boston from the 1970 census.

`medv`	median value of owner-occupied homes in 1000USD
`crim`	per capita crime rate by town
`nox`	nitric oxides concentration (parts per 10 million)
`rm`	average number of rooms per dwelling
`age`	proportion of owner-occupied units built prior to 1940
`ptratio`	pupil-teacher ratio by town

We start by performing some exploratory analysis of the data, which are useful in determining the range and concentration of the covariates and whether there exists any obvious outliers to the dataset.

```
> summary(housing) #quartiles, minimum and mean values of measured variables
> cor(housing)  # pairwise correlation of measured variables
> pairs(housing)  # pairwise scatter plots
```

The `lm` function in `R` is used to fit linear models.

```
> housing.lm1 <- lm(medv ~ crim + nox + rm + age + ptratio, data = housing)
> summary(housing.lm1)
```

Note that we can also write `lm(medv ~., data = housing)`, with the dot indicating that we want to regress `medv` against all other covariates. Make sure you understand the output of `summary(housing.lm1)`.

```
Call:
lm(formula = medv ~ crim + nox + rm + age + ptratio, data = housing)

Residuals:
    Min      1Q  Median      3Q     Max
-6.8922 -1.3853 -0.2392  1.0016  6.9399

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) -38.64308    6.45996  -5.982 3.97e-08 ***
crim         -3.46733    1.14408  -3.031  0.00315 **
nox          18.23441    8.38545   2.175  0.03218 *
rm            9.15886    0.49327  18.568  < 2e-16 ***
age          -0.07390    0.01129  -6.543 3.13e-09 ***
ptratio       0.02124    0.18836   0.113  0.91044
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 2.242 on 94 degrees of freedom
Multiple R-squared:  0.8642,        Adjusted R-squared:  0.857
F-statistic: 119.7 on 5 and 94 DF,  p-value: < 2.2e-16
```

How are all coefficients estimated? How are the standard errors obtained?

Coefficients are estimated by $\hat{\beta} = (X^\top X)^{-1} X^\top Y$ and $\hat{\sigma}^2 = \frac{1}{n-p}\|Y - X\hat{\beta}\|^2$ (note this is the unbiased estimator instead of the MLE), where X is the design matrix and Y the vector of responses. The standard errors are estimated as $se(\hat{\beta}_j) = \hat{\sigma}[(X^\top X)^{-1}]_{jj}^{1/2}$ for $j = 1, \ldots, p$. You may compute these quantities manually using the following code.

```
> X <- model.matrix(~ crim + nox + rm + age + ptratio, data = housing)  # by
default, it includes a column of ones so that the model has an intercept
> Y <- housing$medv
> betahat <- solve(t(X)%*%X, t(X)%*%Y)  # coef estimates
> Yhat <- X%*%betahat
> RSS <- norm(Y - Yhat, type="F")^2
> sigmahat <- sqrt(RSS/(length(Y)-length(betahat)))
> sqrt(diag(solve(t(X)%*%X))) * sigmahat  # standard err
```

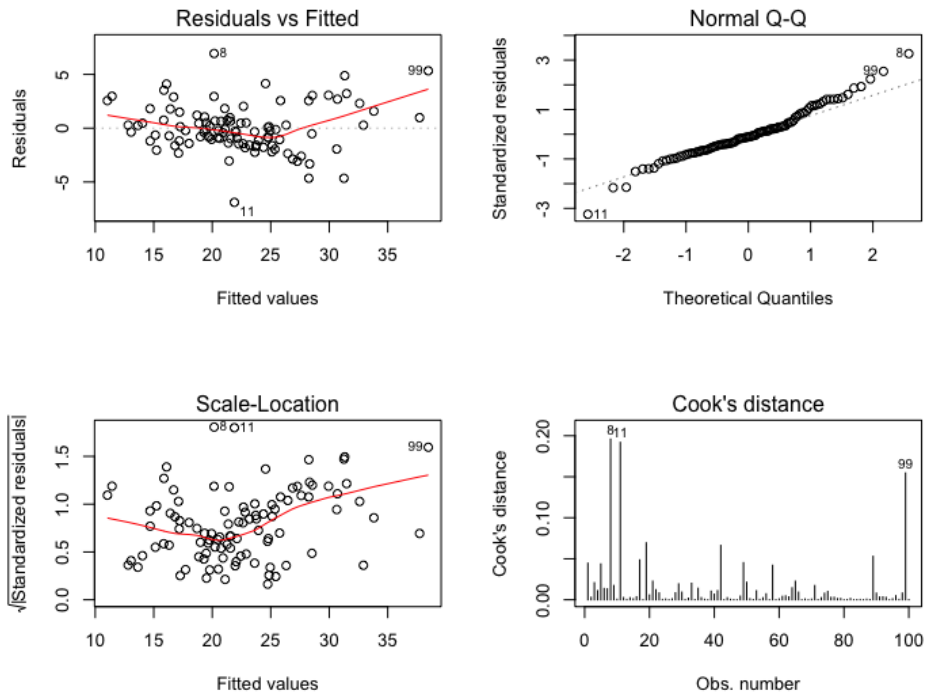
Look at the summary. What is the estimated coefficient of the `rm` variable? *9.16.* What is its interpretation here and how is this different from the coefficient estimate obtained by fitting `lm(medv ~rm, data = housing)`? *Controlling all other covariates, increasing the average number of rooms per dwelling in a region increases its median house valuation by 9160 USD. The coefficient in the univariate model is 10.2, which means the marginal effect of one additional room on the house price is 10224 USD.* What is the meaning of the 'residual standard error' and 'Multiple R-squared' in the output? *The residual standard error is the $\hat{\sigma}$ estimated in the model. Multiple R-squared value of 0.864 means that 86.4% of the variability in the data can be explained by the covariates included in the linear model.* What is the 'F-statistic' in the final line testing? *This is testing the*

intercept-only null model that there is no difference in house prices across the region against the fitted linear model.

Let us now look at some diagnostic plots.

```
> old_par <- par(mfrow = c(2,2))
> plot(housing.lm1, which = c(1,2,3,4))  # diagnostic plots
> par(old_par)
```

Here four plots are generated.



1. Fitted values vs residuals (`which = 1`): under the assumption that $Y_1, \ldots, Y_n$ are independent, the fitted values $\hat{Y}$ and residuals $\hat{\epsilon} := Y - X\hat{\beta}$ are uncorrelated. So we should not expect to see any obvious relationship between fitted values and residuals.

2. Normal Q-Q plot of standardised residuals (`which = 2`): if the linear model assumptions hold, we would expect the vector of residuals $\hat{\epsilon}$ to follow an $N_n(0, \sigma^2(I_n - P))$ distribution, where $P = X(X^\top X)^{-1}X^\top$ is the hat matrix. As a consequence, and due to $\chi^2_m/m \to^d \delta_1$ as $m \to \infty$, where δ_1 is the point mass or Dirac delta at 1, we would expect the standardised residuals $\hat{\epsilon}^*_i := \hat{\epsilon}_i/(\hat{\sigma}\sqrt{1-h_i})$, where $h_i = P_{ii}$ (the leverage of the i^{th} observation), when ordered, to approximately follow the

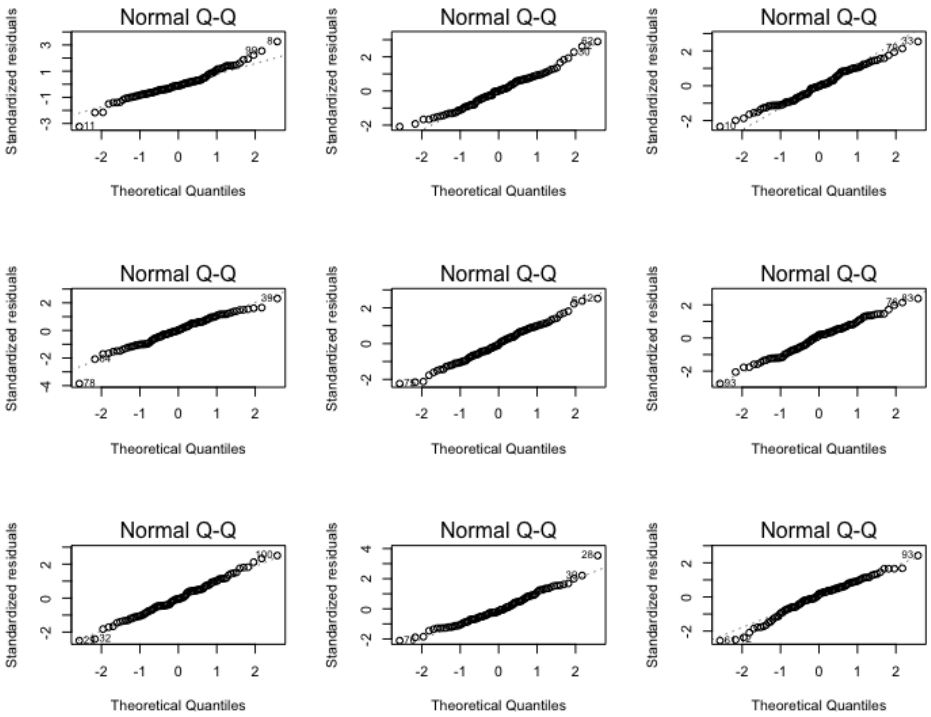
normal quantiles. Higher departure of extreme order statistics from their expected values in Q-Q plots are to be expected.

3. Scale-location plot (`which = 3`): a consequence of the homoscadesticity assumption is that when we plot the fitted values against the square root modulus of the standardised residuals, there should not be a flat trend. Any strong trend (typically upwards) is indicative of heteroscadesticity. This can also be seen in the fitted values vs residuals plot as a funnelling effect.

4. Cook's distance plot (`which = 4`): Cook's distance of the i^{th} observation is defined as $D_i := \frac{1}{p}\|X(\hat{\beta}_{(-i)} - \hat{\beta})\|^2/\hat{\sigma}^2 = \frac{1}{p}\frac{h_i}{1-h_i}(\hat{\epsilon}_i^*)^2$, where $\hat{\beta}_{(-i)}$ is the MLE for β from all observations but i. Large D_i implies that the associated data point has a strong influence on the estimated coefficients. Heuristically, one would go back to the dataset to check if a point is an outlier if its Cook's distance is above $F_{p,n-p}(0.5)$. However, it is typically a bad idea to tamper with data simply because it does not follow the model we want to fit. Only consider removing an outlier if there is suitable scientific justification for doing so.

How do we determine if the diagnostic plots are abnormal? One way is to generated some synthetic Y values from the fitted model and see what the corresponding diagnostic plots look like when the model is true.

```
> old_par <- par(mfrow = c(3,3))  # simultaneously display 9 plots
> plot(housing.lm1, which = 2)  # Q-Q plot from the real dataset
> for (iter in 1:8){
+     # Eight Q-Q plots when the model assumption is true
+     eps = rnorm(dim(housing)[1]) * summary(housing.lm1)$sigma
+     plot(lm(fitted.values(housing.lm1) + eps ~
+                 crim + nox + rm + age + ptratio,data = housing), which = 2)
+ }
> par(old_par)  # revert back to displaying a single plot on the screen
```



The above code generates nine Q-Q plots, including the diagnostic plot from our fitted model `housing.lm1` and eight plots from synthetic data that we know to follow the linear model. Imagine now that someone presents you with these nine plots, randomly shuffled. If you cannot immediately spot the plot from the real data, then there is no clear abnormality in the diagnostic plot.

Insurance data

The data in the file `insurance` shows the number of claims during the last financial year and the ages of 35 customers for a particular health insurance plan. The insurance company wants to investigate how the number of claims is related to a customer's age. Read the data into `R` and visualise it with a scatter plot:

```
> fileName <- "insurance.txt"
> insurance <- read.table(paste0(filePath, fileName), header = TRUE)
> head(insurance)
> with(insurance, plot(age, nclaims))
```

If Y_i and x_i are the number of claims and the age for customer $i = 1, \ldots, 35$, suppose that the Y_is are independent Poisson random variables, with $\mathbb{E}(Y_i) = \mu_i$. Fit the GLM given by $\log \mu_i = \beta_1 + \beta_2 x_i$, $i = 1, \ldots, 35$:

```
> ins.glm1 <- glm(nclaims ~ age, family = poisson, data = insurance)
> ins.glm2 <- glm(nclaims ~ 1, family = poisson, data = insurance)
> ins.glm1
```

The `R` function for fitting a generalised linear model is `glm`. The exponential dispersion family is specified using the `family` option. By default, the canonical link is used for each family. You can specify custom link functions via, e.g., `family = poisson(link = "identity")`; see `?glm` and `?family` for more details.

How are the coefficients $\beta = (\beta_1, \beta_2)^\top$ computed? Recall from lecture that the MLE $\hat{\beta}$ for Poisson regression solves the score equations

$$\nabla \ell(\hat{\beta}) = \sum_{i=1}^{n} x_i (Y_i - e^{x_i^\top \hat{\beta}}) = 0,$$

where x_i is the ith row vector of the design matrix and Y_i the ith response. Though simple, this equation does not admit a closed form solution. Instead, it is typically solved (e.g. in `R`) using Newton–Raphson's method:

1. $\beta^{(0)} = 0$ (or some other initial value)
2. For $t = 1, 2, \ldots$, update $\beta^{(t)} \leftarrow \beta^{(t-1)} - (\nabla^2 \ell(\beta^{(t-1)}))^{-1} \nabla \ell(\beta^{(t-1)})$.

Note that, in a generalised linear model, $\nabla^2 \ell$ may depend on the data and that it is not guaranteed to be invertible at every value of the iteration. Thus, we may substitute it by its expected value, which equals the negative Fisher information matrix under the so-called regularity assumptions (satisfied herein) and which is non-singular. This modification is known as *Fisher Scoring*. We can manually implement it using the code below.

```
> X <- model.matrix(~age, data=insurance)
> Y <- insurance$nclaims
> betahat <- rep(0,2)
> for (iter in 1:5){  # using 5 Fisher scoring iterations
+     score <- t(X) %*% (Y - exp(X%*%betahat))
+     infomat <- t(X)%*%diag(as.vector(exp(X%*%betahat)))%*%X
+     betahat <- betahat - solve(-infomat, score)
+ }
> betahat
```

Look at the summaries and analysis of deviance for these models,

```
> summary(ins.glm1)
> summary(ins.glm2)
> anova(ins.glm2, ins.glm1, test = "Chisq")
```

and interpret them. In particular:

Question. (i) Does the Poisson model give a reasonable fit to the data?

The residual deviance of 45.1 is not statistically significant at level 0.05 when compared to a χ^2_{33} distribution because the p-value, computed as either

```
> 1-pchisq(insurance.glm1$deviance,insurance.glm1$df.residual)
> 1-pchisq(45.062,33)
```

is 0.08. Thus, the Poisson fit is reasonable.

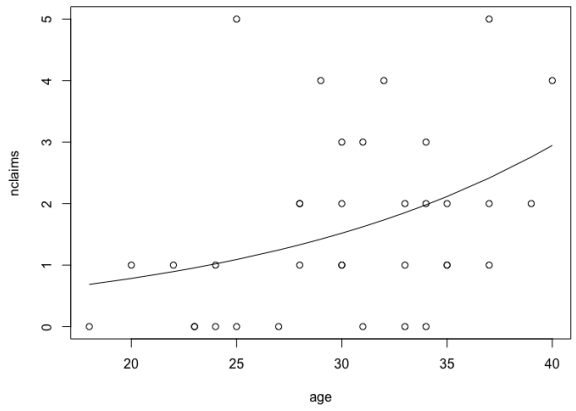
(ii) Is the drop in deviance due to including β_2 statistically significant at significance level 0.05? *Inclusion of the* `age` *covariate decreases the deviance by 6.76, which is significant compared to a χ^2_1 distribution (p-value $<$ 0.01$<$0.05).*

(iii) What is $\hat{\beta}_2$ and what is an approximate 95% confidence interval for β_2? *$\hat{\beta}_2 = 0.0663$ and the confidence interval has end points $0.0663 \pm 1.96 \times 0.0262$.*

(iv) What is the estimated effect of one year's increase in age on the average number of claims? Give an approximate 95% confidence interval for this effect. *One year's increase in age increases the average number of claims by $\exp(\hat{\beta}_2)$ and the confidence interval for this effect has endpoints approximately equal to $\exp(0.0663 \pm 1.96 \times 0.0262)$.*

Illustrate your fit graphically.

```
> points(insurance$age, ins.glm1$fitted.values, type="l")
```



Let us now look at some residual plots. Note that the raw residuals are no longer useful, as their variances are not constant. Two more popular type of residuals in generalised linear models are Pearson and deviance residuals:

$$e_i := \frac{Y_i - \hat{\mu}_i}{\sqrt{a_i V(\hat{\mu}_i)}} \quad \text{and} \quad d_i := \text{sign}(Y_i - \hat{\mu}_i)\sqrt{\frac{2}{a_i}\big(\theta(Y_i)Y_i - K(\theta(Y_i)) - [\theta(\hat{\mu}_i)Y_i - K(\theta(\hat{\mu}_i))]\big)}.$$

Clearly, $(n-p)^{-1}\sum_{i=1}^{n} e_i^2 = \hat{\phi}$ and $\sum_{i=1}^{n} d_i^2 = D(Y;\hat{\mu})$. Let

$$H(\mu) := W^{1/2}(\mu)X\left(X^\top W(\mu)X\right)^{-1}X^\top W^{1/2}(\mu)$$

and let $h_i = H_{ii}(\mu)$ (again, the leverage of the i^{th} observation). Then, as $\phi \to 0$,

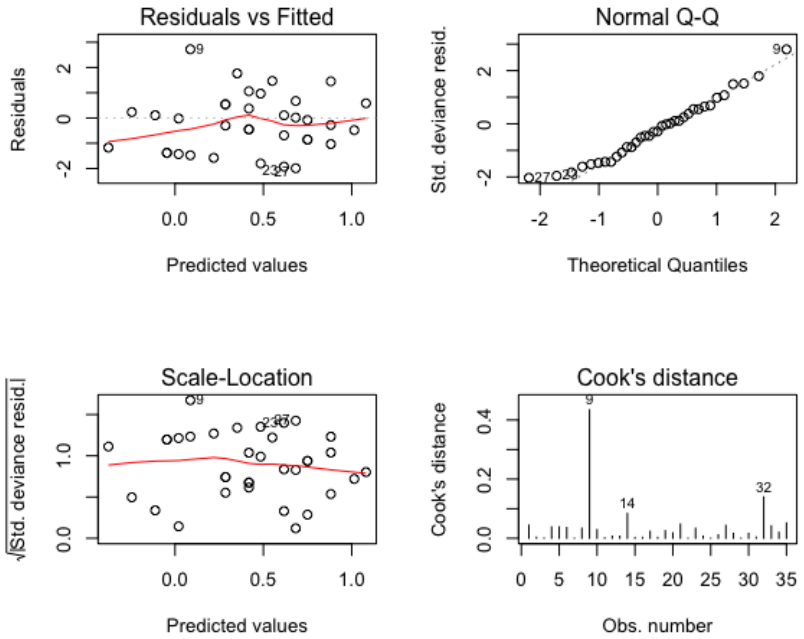
$$\frac{1}{\sqrt{\phi}}(e_1,\ldots,e_n), \frac{1}{\sqrt{\phi}}(d_1,\ldots,d_n) \to^d N(0, I-H)$$

if ϕ known and, if ϕ is unknown,

$$\frac{1}{\hat{\phi}^{1/2}}(d_1,\ldots,d_n) \to^d t_{n-p}(0, I-H),$$

so we can use Q-Q plots with the standarised residuals $e_i^* := e_i/\sqrt{1-h_i}$ and $d_i^* := d_i/\sqrt{1-h_i}$. Also, plotting the d_i^*s versus the linear predictors or predicted values $g(\hat{\mu}_i)$ should not reveal any obvious trend. Finally, we still define Cook's distance as $D_i := \frac{1}{p}\frac{h_i}{1-h_i}(d_i^*)^2$. Run the following commands and convince yourself that pretty much the same rules apply as for ordinary linear models.

```
> old_par <- par(mfrow=c(2,2))
> plot(ins.glm1, which=c(1,2,3,4))
> par(old_par)
```



Practical 2: Logistic regression and model selection

Seeds data

The data show the results of an experiment to investigate the effect of seed variety (types A and B) and root extract (types 1 and 2) on germination of seeds.

The data are `y`, the number of seeds germinated, out of a total `n` sown, so that of 39 type A seeds with root extract 1, there were 10 seeds that germinated. First read in the data.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "seeds.txt"
> seeds <- read.table(paste0(filePath, fileName), header=T)
> head(seeds)

##   seed extract  y  n
## 1    A       1 10 39
## 2    A       1 23 62
## 3    A       1 23 81
## 4    A       1 26 51
## 5    A       1 17 39
## 6    A       2  5  6
```

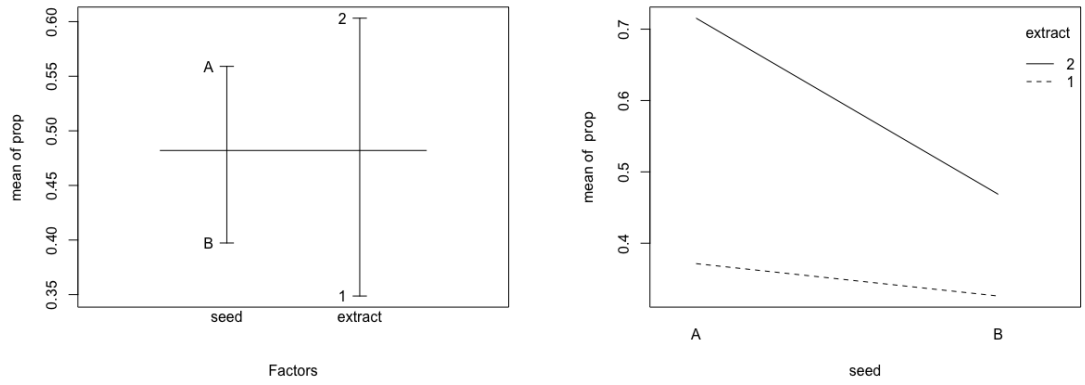
To understand the effect of seed variety and root extract on germination, we may use a binomial regression model. In `R`, the value of the dependent variable in this model must be between 0 and 1. Thus, we need to set up the data for modelling by defining a new variable `prop` as follows.

```
> seeds <- cbind(seeds, prop = seeds$y/seeds$n)
> is.factor(seed)  # check for 'extract' too
> seeds$extract  <- as.factor(seeds$extract)  # if not already factor
> head(seeds)

##   seed extract  y  n      prop
## 1    A       1 10 39 0.2564103
## 2    A       1 23 62 0.3709677
## 3    A       1 23 81 0.2839506
## 4    A       1 26 51 0.5098039
## 5    A       1 17 39 0.4358974
## 6    A       2  5  6 0.8333333
```

Now let us do some exploratory plots for factor designs (models with categorical variables).

```
> attach(seeds)  # what does 'attach' do?
> plot.design(prop ~ seed*extract) # if 'attach' not used, add argument 'data=seeds'
> interaction.plot(seed, extract, prop)
```



How do you interpret these plots? *The first simply suggests that, marginally, seed A and extract 2 make germination more likely. The interaction plot suggest that there is a strong interaction (lines are not parallel), and that whilst seed A is better than seed B, the effect is only strong under extract 2. In other words, it seems to be the combination of seed A and extract 2 which is beneficial. Note, however, that the proportions are not weighted by sample size in either plot, so these conclusions are not necessarily true!*

Let us fit a binomial regression model to this data, taking into account the sample size for each proportion and including the interaction between the factors.

```
> seed.glm1 <- glm(prop ~ seed*extract, family=binomial, weights=n, data = seeds)
> summary(seed.glm1)

## Call:
## glm(formula = prop ~ seed * extract, family = binomial, data = seeds,
##     weights = n)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.0162  -1.2440   0.0599   0.8470   2.1212
##
## Coefficients:
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -0.5582     0.1260  -4.429 9.46e-06 ***
## seedB            0.1459     0.2232   0.654   0.5132
## extract2         1.3182     0.1775   7.428 1.10e-13 ***
## seedB:extract2  -0.7781     0.3064  -2.539   0.0111 *
## ---
## Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 ##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 98.719  on 20  degrees of freedom
## Residual deviance: 33.278  on 17  degrees of freedom
## AIC: 117.9
##
## Number of Fisher Scoring iterations: 4
```

Note that in the first line of code we set `weights=n`. This groups up data points with same `seed` and `extract` values by computing the weighted sum (according to their values of `n`) of their `prop` values. Make sure you are convinced that this is sensible for this model.

Write down the algebraic form of the model that has been fitted. *The model fitted is*

$$\mathit{prop}_{ij} = \sum_{k=1}^{K_{ij}} w_{ijk}\mathit{prop}_{ijk} \sim^{ind.} \frac{1}{N_{ij}} \text{Bin}\left(N_{ij}, p_{ijk}\right),$$

where $w_{ijk} = \mathit{n}_{ijk}/N_{ij}$, $N_{ij} = \sum_{l=1}^{K_{ij}} \mathit{n}_{ijl}$ *and* $\text{logit}(p_{ijk}) = \log(p_{ijk}/(1-p_{ijk})) = \mu+\alpha_i+\beta_j+\gamma_{ij}$, *for seed type* $i = 1, 2$ *(corresponding to* `A` *and* `B`*), extract* $j = 1, 2$ *and experiments* $k = 1, \ldots, K_{ij}$, *subject to corner-point constraints* $\alpha_1 = \beta_1 = \gamma_{11} = \gamma_{12} = \gamma_{21} = 0$.

Is the model a good fit?

The model has a residual deviance of 33.3 on 17 degrees of freedom, and if we compare 33.3 to a χ^2_{17} *we get a p-value of 0.0103, which is small and suggests a poor fit. In other words, there is probably some other factor affecting the seeds which we haven't measured, or the logistic model is simply not correct.*

Can you interpret the model coefficients? *The MLEs are* $\hat{\mu} = -0.56$, *which is the logit of the fitted probability (or log-odds) of germination of seed A under extract 1;* $\hat{\alpha}_2 = 0.15$

is the change (increase) in the log-odds of germination obtained by using seed B instead of seed A with extract 1; $\hat{\beta}_2 = 1.3$ is the chance (increase) in log-odds for using extract 2 instead of extract 1 with seed A; $\hat{\gamma}_{22} = -0.778$ is the additional change (decrease) from using the combination of extract 2 with seed B over their individual effects.

What are the estimated probabilities of germination under the various growing conditions? *Just use the expit function to invert the log-odds. Probabilities for seed A are 0.364 and 0.681 under extracts 1 and 2 respectively, and for seed B they are 0.398 and 0.532 respectively. Thus, we see that by taking into account the weights (and according to this model) the mean proportions partly disagree with those shown in the interaction plot above.*

```
> anova(seed.glm1, test="Chisq") # to get chisq p-values

## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: prop
##
## Terms added sequentially (first to last)
##
##
##              Df Deviance Resid. Df Resid. Dev Pr(>Chi)
## NULL                             20     98.719
## seed          1    2.544        19     96.175  0.11070
## extract       1   56.489        18     39.686 5.65e-14 ***
## seed:extract  1    6.408        17     33.278  0.01136 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

What are the chi-squared p-values testing here? *Under the algebraic form above, the three p-values correspond respectively to testing $H_0 : \text{logit}(p_{ijk}) = \mu$ against $H_1 : \text{logit}(p_{ijk}) = \mu + \alpha_i$ (p-value = 0.11), testing H_1 against $H_2 : \text{logit}(p_{ijk}) = \mu + \alpha_i + \beta_j$ (p-value < 0.001) and testing H_2 against $H_3 : \text{logit}(p_{ijk}) = \mu + \alpha_i + \beta_j + \gamma_{ij}$ (p-value = 0.01). In particular, all the p-values are computed using chi-squared distributions on 1 degree of freedom.*

Let us fit a new model.

```
> seed.glm2 <- glm(prop ~ seed+extract, family=binomial, weights=n, data = seeds)
```

Check the difference in deviances to test whether the interaction term is needed. (Does this make sense after our conclusion above?) *The difference in deviances is 6.41, which is large compared to a χ^2_1 under small dispersion asymptotics (note that we could have noticed this directly from the output of `anova` above). We conclude therefore that the simpler model is rejected and the more complex model is preferred. This conclusion still makes some sense even if the complex model is itself a poor fit.*

Surgery data

Data are collected for 40 patients receiving a new surgery technique for tumour removal, The variable `death` takes the value 1 if the patient dies within 30 days of surgery and is 0 otherwise. Other covariates recorded include age of patient in years (`age`), sex of patient (`sex`), whether vascular invasion is present (`vas`), the natural logarithm of maximum diameter (in mm) of tumours removed (`lnSize`), clinical staging of the tumour (`stage`) and platelet level in 1000s per microlitre before surgery (`platelet`). Notice that the response variable `death` is **binary** data, a special case of binomial data.

```
> fileName <- "surgery.txt"
> surg <- read.table(paste0(filePath, fileName), header=T)
> head(surg)
#   age sex vas lnSize stage platelet death
# 1  50   M   1   4.25     3      214     1
# 2  50   F   0   2.71     2      135     0
# 3  51   M   1   2.94     1      270     1
# 4  51   M   0   3.50     2      293     0
# 5  53   M   1   4.09     3      209     1
# 6  54   M   0   2.30     1      182     0
> attach(surg)
> sum(death)  # what does this tell you?
```

The question of interest is to find predictive factors for post-operative survival. The `stage` variable is ordinal (categorical with ordering). By default, logistic regression cannot handle ordinal variables in the sense that the fitted model does not necessarily respect their natural order. We can treat `stage` as a continuous variable if we believe the difference in survival (at log-odds level) between successive stages are the same. However, in this case, we will assume it is clinically reasonable to group Stages 1/2 and Stages 3/4 together, so that we convert `stage` to a two-level categorical variable that regards Stages 1/2 as the base-level, and the estimated parameter associated to the new `stage` variable is likely to be positive so that the model respects the natural order. We can fit a logistic model as follows (`R` will assume `weights` are all equal if we don't specify).

```
> surg.glm1 <- glm(death~age+sex+vas+lnSize+I(stage>=3)+platelet, binomial)
> summary(surg.glm1)
```

Interpret the `R` output for this model. Can you summarise the effect of maximum tumour diameter on the odds of dying within 30 days? *The coefficient for `lnSize` is 2.25. Hence every one unit increase in the logarithm of the maximum tumour diameter increases the log-odds of dying (i.e. logit of the probability of dying) within 30 days by 2.25, controlling for all other covariates (i.e. this is not a marginal effect).*

We can use AIC to select a best predictive model among all possible submodels. However, when the number of these is large this may be prohibitively expensive. Here, $p = 6$ so

we have $2^{6-1} = 32$ submodels (generally we choose to include the intercept in all). Instead, and as it is usually done when p is large, we employ a greedy approach: we start from an initial model and, in each step, explore the model space by picking the best model among all models that are 'close' to the last explored model according to some rule; lastly, we choose among all the selected models according to AIC, BIC or cross validation, depending on the purpose. Two popular strategies are the following (we introduce them within the context of generalised linear models). Assume we have data $Z = (x_1, Y_1), \ldots, (x_n, Y_n)$ and let

$$\mathcal{M}^{(S)} := \left\{ \prod_{i=1}^{n} P_{\mu_i^{(S)}, \phi_i} : \mu_i^{(S)} = \mu\left(\left(x_i^{(S)}\right)^\top \beta^{(S)} \right), x_i^{(S)} = (x_{ij})_{j \in S}, \beta^{(S)} = (\beta_j)_{j \in S}, \phi_i = a_i \phi \right\},$$

where $S \subseteq \{1, \ldots, p\}$. Let $L(Z, \mathcal{M})$ be some discrepancy measure between Z and $\mathcal{M}$; e.g., the deviance $D(Y; \hat{\mu})$, AIC, BIC or cross-validation.

Forward selection

1. Set $S_1^* = \{1\}$ if an intercept is included (placed, w.l.o.g., at the first column of the design matrix) or, otherwise, set

$$S_1^* = \underset{\substack{S = \{j\}: \\ j \in \{1, \ldots, p\}}}{\arg\min} L(Z, \mathcal{M}^{(S)});$$

2. for $j = 2, \ldots, \min\{p, n\}$, set

$$S_j^* = \underset{\substack{S = S_{j-1}^* \cup \{k\}: \\ k \in \{1, \ldots, p\} \setminus S_{j-1}^*}}{\arg\min} L(Z, \mathcal{M}^{(S)}); \text{ and}$$

3. select $\mathcal{M}^{(S^*)}$ from $\{\mathcal{M}^{(S_1^*)}, \ldots, \mathcal{M}^{(S_{\min\{p,n\}}^*)}\}$ using AIC, BIC or cross-validation.

Backward selection Assume $p \leq n$. Then,

1. set $S_1^* = \{1, \ldots, p\}$;
2. for $j = 2, \ldots, p$, set

$$S_j^* = \underset{\substack{S = S_{j-1}^* \setminus \{k\}: \\ k \in S_{j-1}^*}}{\arg\min} L(Z, \mathcal{M}^{(S)}); \text{ and}$$

3. select $\mathcal{M}^{(S^*)}$ from $\{\mathcal{M}^{(S_1^*)}, \ldots, \mathcal{M}^{(S_p^*)}\}$ using AIC, BIC or cross-validation.

We use forward selection (or stepwise greedy algorithm) with AIC (both for the search strategy and the selection criterion) to select a submodel.

```
> step(glm(death~1,binomial), scope=death~age+sex+vas+lnSize+I(stage>=3)+platelet,
direction="forward", trace=1)
```

```
Start:  AIC=54.93
death ~ 1

                Df Deviance    AIC
+ I(stage >= 3)  1   33.364 37.364
+ lnSize         1   37.426 41.426
+ vas            1   47.527 51.527
+ sex            1   49.568 53.568
<none>               52.925 54.925
+ age            1   52.829 56.829
+ platelet       1   52.863 56.863

Step:  AIC=37.36
death ~ I(stage >= 3)

           Df Deviance    AIC
+ lnSize    1   27.278 33.278
<none>          33.364 37.364
+ vas       1   32.122 38.122
+ age       1   32.552 38.552
+ sex       1   33.081 39.081
+ platelet  1   33.356 39.356

Step:  AIC=33.28
death ~ I(stage >= 3) + lnSize

           Df Deviance    AIC
<none>          27.278 33.278
+ age       1   26.647 34.647
+ sex       1   27.218 35.218
+ vas       1   27.219 35.219
+ platelet  1   27.231 35.231

Call:  glm(formula = death ~ I(stage >= 3) + lnSize, family = binomial)

Coefficients:
(Intercept)  I(stage >= 3)TRUE             lnSize
-8.180              3.394              2.123

Degrees of Freedom: 39 Total (i.e. Null);  37 Residual
Null Deviance:     52.93
Residual Deviance: 27.28  AIC: 33.28
```

Look at the output. What does the `step` function do? *It starts from the intercept-only null model, and at each step greedily adds the covariate that minimises the AIC when*

included to the previous model. In this case, the covariates added are `I(stage >= 3)` *and* `lnSize`.

What are the fitted parameters in the chosen model? *The final fitted model is*

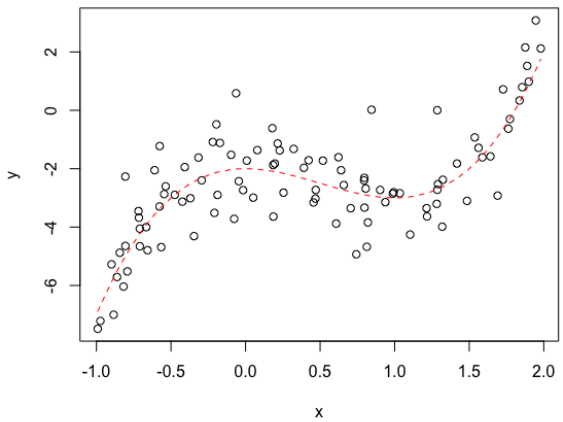
$$\text{logit}\,\mathbb{P}(\texttt{death}) = -8.18 + 3.39\,\mathbb{1}(\texttt{stage} >= 3) + 2.12\,\texttt{lnSize}.$$

Are confidence intervals obtained via `confint(glm(death ~I(stage>=3)+lnSize, family=binomial))` valid? *No, because any inferential procedures computed after and from the same data used for model selection are not valid. And even if we attempt to give them a frequentist meaning by looking at the frequencies when the selected model is selected again if we can run the experiment many times, the resulting confidence intervals will be overly optimistic (as in, too narrow by not taking into account the uncertainty from selecting the model).*

Model selection

We now look at various model selection methods in a synthetic dataset. We generate 100 pairs (x, y) where the true model assumes $Y_i = -2 - 3x^2 + 2x^3 + \epsilon_i$ for $\epsilon_i \overset{\text{iid}}{\sim} N(0, 1)$.

```
> set.seed(2018) # so that we all get the same 'random' numbers
> n = 100
> x = sort(runif(n,-1,2))
> mu = -2 - 3*x^2 + 2*x^3
> y = mu + rnorm(n)
> plot(x,y)
> points(x,mu,type="l")
```

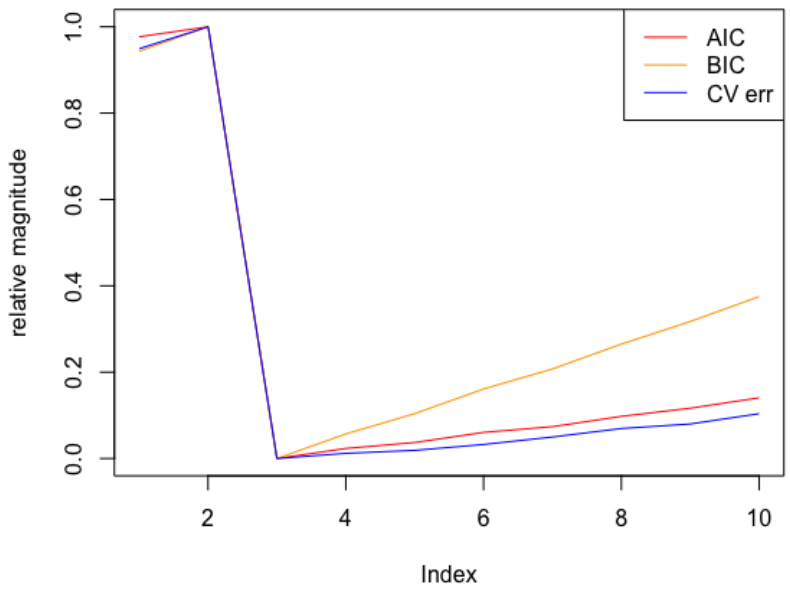


We fit polynomials of various degrees (1 to 10) to the data. We use the following function to calculate V-fold cross-validation error. Do you understand what is done in the `for` loop? (The `[-ind]` notation is used to extract all indices except those included in `ind`.)

```
> VfCV <- function(x, y, degree, folds){
+     n = length(x)
+     yhat <- rep(0, n)
+     foldID <- (rep(0, n) + 1:folds)[sample(n)]
+     for(v in 1:folds) {
+         ind <- (1:n)[foldID == v]
+         x.train <- x[-ind]
+         x.test <- x[ind]
+         y.train <- y[-ind]
+         fit <- lm(y.train ~ poly(x.train, degree=degree) )
+         yhat[ind]<- predict(fit, newdata=data.frame(x.train = x.test))
+     }
+     # the squared error:
+     return(sum((y-yhat)^2))
+ }
```

The `shape` function below is used to linearly rescale AIC, BIC and cross-validation errors so that they fit on the same plot.

```
> shape <- function(x){(x-min(x))/(max(x)-min(x))}
> AICs <- BICs <- VfCVerr <- rep(0, 10)
> for (degree in 1:10){
+     fit <- lm(y ~ poly(x, degree))
+     AICs[degree] = AIC(fit)
+     BICs[degree] = BIC(fit)
+     VfCVerr[degree] = VfCV(x, y, degree, folds=n)
+ }
>
> plot(shape(AICs), type="l", col="red", ylab="relative magnitude")
> points(shape(BICs), type="l", col="orange")
> points(shape(VfCVerr), type="l", col="blue")
> legend("topright", c("AIC","BIC","VfCV err"), lty=1, col=c("red","orange","blue"))
> which.min(AICs)
> which.min(BICs)
> which.min(VfCVerr)
```



We see that all three method select the cubic polynomial model. The BIC gives a sharper minimum at the degree 3 model. Moreover, they show much larger penalty for under-fitting (degree less than 3) than over-fitting (degree larger than 3). This is because in this polynomial fitting example, under-fitting typically result in large increase in residual sums of squares, whereas over-fitting at most increase AIC by 1 (or BIC by $\log n$) per additional parameter. Also, we see that the cross-validation errors and AIC show very similar behaviour across the ten models. This is in line with the asymptotic equivalence of these two methods.

Practical 3: Regularised regression

This practical is about how to fit generalised linear models with ridge or Lasso regularisations using the `glmnet` package. Install and load the package first:

```
install.packages("glmnet")
library(glmnet)
```

Prostate cancer data

This dataset is in the `ElemStatLearn` package, which is no longer available for new versions of `R`. You may still find it on the website of the book:

```
> website <- "https://web.stanford.edu/~hastie/ElemStatLearn/datasets/prostate.data"
> prostate <- read.table(website, header=T)
```

It looks at how a clinical indicator of prostate cancer `lpsa` depends on patient characteristics and levels of a number of prostate-specific antigens. Load the dataset and perform some exploratory analysis.

```
> str(prostate) # structure of data
> cor(prostate[,1:8])
> pairs(prostate[,1:9], col="blue")
```

Observations in the `prostate` dataset have a logical attribute `train`. About two thirds of the data would be used for training our models (`train==TRUE`) and the remaining data held for testing (`train==FALSE`).

```
> train <- subset(prostate, train == TRUE)[,1:9]
> test <- subset(prostate, train == FALSE)[,1:9]
```

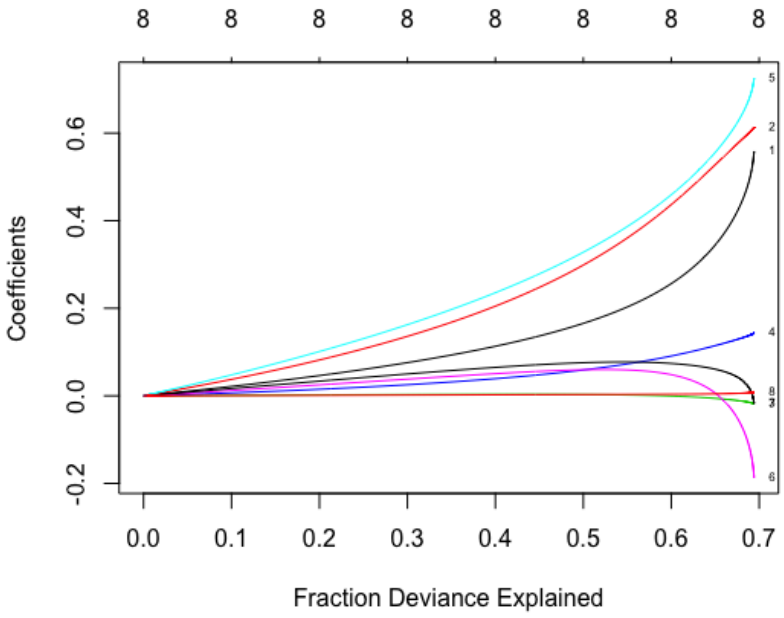
Ridge regression. The `glmnet` function can be used to fit both ridge and Lasso regressions (or a convex combination of the two penalties, known as an *elastic net*, hence the name of the package, see `?glmnet` for more details). For ridge regression, we need to set the parameter `alpha = 0`. Unlike `lm` and `glm` functions, `glmnet` does not take formulae as input for a model. We have to specify a numerical design matrix and a response vector instead. Note that this means logical or factor variables in the data frame have to be converted to numerical variables. The function `data.matrix` does it for us (this is very similar to `model.matrix` function, but does not append a column of 1s for intercept. `glmnet` will add in the intercept for us).

```
> X <- data.matrix(train[,1:8])
> Y <- train$lpsa
> prostate.ridge <- glmnet(X,Y,family = "gaussian", alpha=0,
lambda.min.ratio=1e-6,nlambda=1000)
```

Note that since the ridge regression estimator $\hat{\beta}_{\lambda}^{\text{ridge}}$ is indexed by the regularisation parameter λ, function `glmnet` by default solves a whole sequence of regression problems for a range of values of λ. By default, `glmnet` starts from $\lambda = \lambda_{\max}$, chosen just large enough such that the estimated parameters are identically zero. Since in ridge regression no (finite) value of λ achieves this, `glmnet` chooses $\lambda_{\max}$ as the smallest value at and after which the estimated parameters are identically zero for `alpha = 0.001`. Then, it solves the ridge regression for a grid of `nlambda` values of λ equally spaced on the logarithmic scale, with the minimum λ value defined to be `lambda.min.ratio` times $\lambda_{\max}$. We can alternatively specify our own sequence of λ values (or even just one specific λ) using the `lambda` argument. Though for optimal computational speed, the default setup is preferred.

The simplest way to visualise the output of `glmnet` is by plotting the path of ridge regression estimators.

```
> plot(prostate.ridge, xvar="dev", label=TRUE)
```

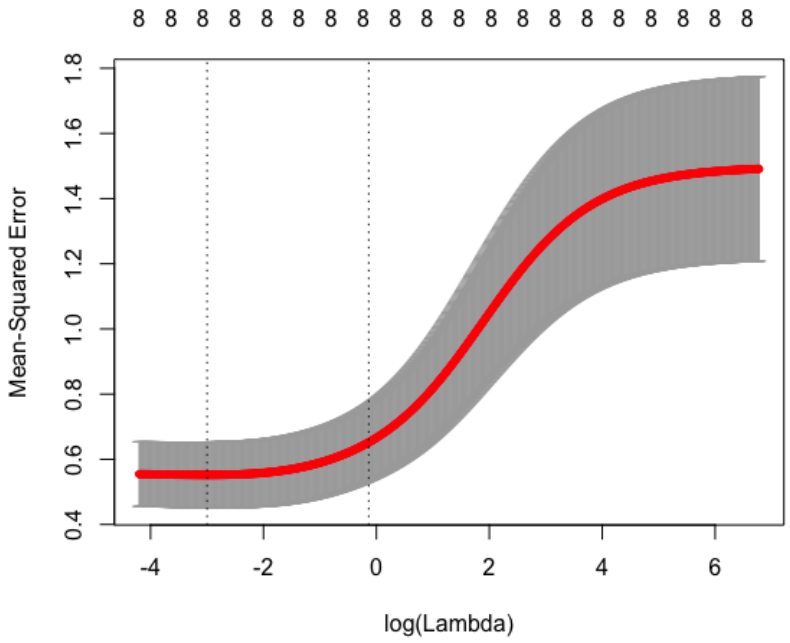


The horizontal axis measures the fraction of deviance explained by the fitted model; so the model complexity increases as we move to the right. The coloured curves show

how the coefficient estimates for different covariates evolve through the sequence of λ values. Try `xvar="lambda"` to plot coefficients against $\log \lambda$ instead (this is called the regularisation path).

Cross validation. We choose the best λ via cross-validation. In this case, the `glmnet` package has implemented a cross validation procedure in the `cv.glmnet` function. By default, it uses the deviance as the loss function.

```
> set.seed(2018) # for reproducible random partitions into V folds
> prostate.cvridge <- cv.glmnet(X,Y, family = "gaussian",alpha=0,
lambda=prostate.ridge$lambda,nfolds=10)
> prostate.cvridge$lambda.min  # optimal regularisation parameter
> plot(prostate.cvridge)
```



The red curve shows the average cross validation error in the K folds ($K = 10$ in this case). The cross validation error decreases and then quickly increases, indicating that a relatively small regularisation parameter should be used. The optimal choice of λ is indicated by the first vertical dotted line (the second corresponds to the largest value within 1 standard error of the optimal λ, `prostate.cvridge$lambda.lse`, which hedges against overfitting). What are the estimated parameters for this choice of λ? [Hint: look up the values of the functions `glmnet` and `cv.glmnet`.] How do you interpret the coefficient for `age` in the chosen model? *We can extract the coefficient estimates from `prostate.ridge` using the following commands.*

```
> ind <- prostate.ridge$lambda==prostate.cvridge$lambda.min
> round(prostate.ridge$a0[ind], 3)  # intercept
s701
0.19
> round(prostate.ridge$beta[,ind], 3)  # coefficients
 lcavol lweight     age    lbph     svi     lcp gleason   pgg45
  0.520   0.607  -0.016   0.141   0.699  -0.145   0.004   0.008
```

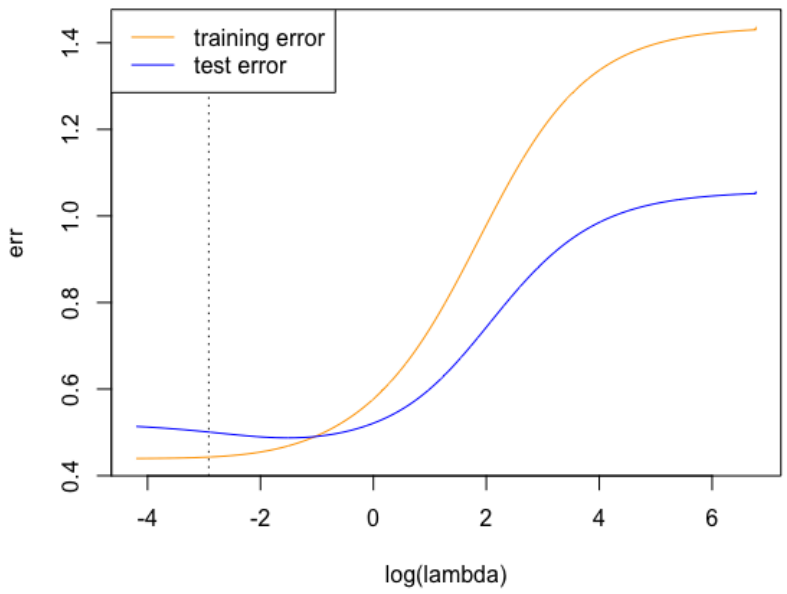
Hence[1] *the intercept is 0.19, the parameter for* `lcavol` *is 0.520, etc. The coefficient for* `age` *is -0.016, showing that every one year increase in age decreases the* `lpsa` *level by 0.016 units, all the rest of covariates being fixed. Note that by default,* `glmnet` *will standardise columns of* X *before performing ridge regression, so as to make every column unitless. However, the estimated coefficients are then transformed back. So we can interpret the coefficients as usual with the original data matrix (and the associated units).*

We remark that only point estimates are available in `glmnet`. It is a very natural question to ask for standard errors of regression coefficients or other estimated quantities. This package deliberately does not provide standard errors. The reason for this is that standard errors are not very meaningful for strongly biased estimates such as arise from regularised estimation methods. Recall that regularised estimation is a procedure that reduces the variance of estimators by introducing substantial bias. The bias of each estimator is therefore a major component of its mean squared error, whereas its variance may contribute only a small part. De-biasing the penalised regression estimators and constructing confidence intervals is still an active field of research.

Training vs test errors. In our case, as we have set aside some testing data, how does the cross-validation choice of λ compare to the optimal λ that minimises the test error?

```
> Yhat <- predict(prostate.ridge, X) # computed for all values of lambda
> train_err <- colMeans((Y - Yhat)^2)
>
> X_new <- data.matrix(test[,1:8])
> Y_new <- test$lpsa
> Y_newhat <- predict(prostate.ridge, X_new)
> test_err <- colMeans((Y_new - Y_newhat)^2)
>
> plot(log(prostate.ridge$lambda), train_err, col="orange", type="l",
+      ylab="err", xlab="log(lambda)")
> points(log(prostate.ridge$lambda), test_err, col="blue", type="l")
> abline(v=log(prostate.cvridge$lambda.min),lty=3)
> legend("topleft", c("training error","test error"),
+        lty=1, col=c("orange","blue"))
```

[1] Typo: the numerical values are now correct, I probably used the wrong seed in the first version.



We note that the training error always decreases as λ decreases, since we always improve our fit with a more complex model. However, the test error first decreases, then increases; this agrees with out intuition since, for prediction, we think that introducing some bias (but not too much) should be good. The dotted line is the location of the optimal regularisation value chosen via cross-validation. It is a bit smaller than the minimum of the test error curve. However, remember that what we see here is only a finite sample approximation of the population test error. So given the relatively flat left tail of the curve, our cross validation choice of regularisation parameter is not too bad.

Lasso regression. Now repeat the exercise using Lasso instead of ridge regression. Recall that `glmnet` fits a Lasso estimator when we choose `alpha=1` in the argument.

```
> prostate.lasso <- glmnet(X,Y,alpha=1,lambda.min.ratio=1e-6,nlambda=1000)
> plot(prostate.lasso, xvar = "dev", label=TRUE)
```

Compare the Lasso path and the ridge path. What do you observe? *Lasso estimators are more sparse, especially when the model complexity is low. We see different variables entering the model discretely as model complexity increases.* What are the cross-validated Lasso coefficients?

```
> set.seed(2018)
> prostate.cvlasso <- cv.glmnet(X,Y,alpha=1,lambda=prostate.lasso$lambda,nfolds=10)
> prostate.cvlasso$lambda.min  # optimal regularisation parameter
> ind <- prostate.lasso$lambda==prostate.cvlasso$lambda.min
> round(prostate.lasso$a0[ind], 3)  # intercept
## 0.166
> round(prostate.lasso$beta[,ind], 3)  # coefficients
##  lcavol lweight     age    lbph     svi     lcp gleason   pgg45
##   0.544   0.596  -0.015   0.135   0.670  -0.146   0.000   0.007
```

Note that `gleason` *was set to zero exactly!*

GWAS data

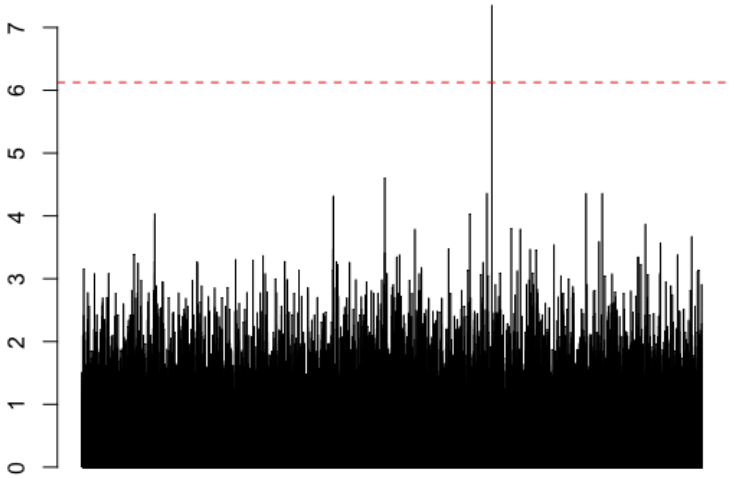
Regularised regression is often used in genome-wide association studies (GWAS), where the number of genetic markers measured greatly exceeds number of patients enrolled in the studies. The following is a GWAS dataset with 89 patients, out of which 39 have a certain disease. Each row of the data contains the following information: patient identifier (`pid`), disease indicator (`disease`) and 66535 columns of single nucleotide polymorphism (SNP) information. An SNP represents a variation in a single nucleotide (A/T/C/G) in a certain position of the genome across the human population. The SNP information in the dataset are represented by 0 for major allele (i.e. common type) and 1 for minor allele (i.e. mutated type).

Read in the dataset from the course website. If you find the file too big for your machine, you can use the smaller dataset `gwas_small` where a subset of 4000 SNPs is included.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "gwas.txt"  # or fileName <- "gwas_small.txt"
> gwas <- read.table(paste0(filePath, fileName), header=TRUE)  # this may take a while
> dim(gwas)
> gwas[1:10,1:10]
```

We can perform single-covariate logistic regression of disease outcome against each SNP and plot the p-values. The resulting plot is known as the Manhattan plot. The computation can be a bit slow, so we use the `printPercentage` function in the `rje` package to monitor progress.:

```
> install.packages("rje")
> library(rje)
> dev <- rep(0,66536)
> for (j in 1:66536){
+     dev[j] <- glm(disease~.,family="binomial",data=gwas[,c(2,2+j)])$deviance
+     printPercentage(j,66536)
+ }
> nulldeviance <- glm(disease~1, family="binomial", data=gwas)$deviance
> pval <- 1 - pchisq(nulldeviance - dev, 1)
> barplot(-log10(pval)) # Manhattan plot
> abline(h=-log10(0.05/66536), col="red",lty=2) # p-val = 0.05/66536
> colnames(gwas)[order(pval)[1:6]+2] # six most significant SNPs (individually)
```



The red dashed line indicates 5% level using Bonferroni correction (i.e. dividing the test level by the number of hypotheses tested). This is needed because we are performing several (66536) tests simultaneously: each test tests that all single covariates are not present in the model versus each of them being present, so we are effectively testing that all single covariates are not present in the model versus that at least one is, and if we wish to achieve significance α at the simultaneous test we should have significance $\alpha/66536$ for each of them.

Do you see any reason why Lasso is preferred over ridge regression in this case? *It is reasonable to believe that only a small number of all genetic markers is associated with a certain disease and/or that we wish to explain the disease by a small number of all genetic markers. Hence, Lasso is preferred over ridge regression in this case.* Use

the `family="binomial"` argument to perform a Lasso penalised logistic regression using `glmnet`.

```
> X <- data.matrix(gwas[,-c(1,2)])
> Y <- gwas$disease
> gwas.lasso <- glmnet(X,Y,alpha=1,family='binomial')
plot(gwas.lasso, xvar="dev", label=TRUE)
```

We again choose the tuning parameter λ via a 10-fold cross-validation.

```
> set.seed(2018)
> gwas.cvlasso <- cv.glmnet(X,Y,family="binomial", alpha=1,
> lambda=gwas.lasso$lambda,nfolds=10)
> plot(gwas.cvlasso)

> gwas.cvlasso$lambda.min
> ind <- gwas.lasso$lambda==gwas.cvlasso$lambda.min
> round(gwas.lasso$a0[ind], 3)
> betahat <- round(gwas.lasso$beta[,ind], 3)
> betahat[betahat!=0]
```

Write down the model chosen from the cross validation. *Only four covariates have non-zero coefficients in the cross-validated model. The chosen model, including the intercept term, is* $Y_i \sim^{ind.} Bin(1, p_i)$, *where* p_i *is the probability that the* i*th individual has the disease and it satisfies*

$$\operatorname{logit} p_i = -0.408 + 0.159x_{i1} + 0.133x_{i2} + 0.087x_{i3} + 0.796x_{i4}$$

for $x_{ij}, j = 1, \ldots, 4$, *the indicators of mutation for the* i*th individual at SNP locations rs9482200_2, rs1119114_1, rs10830072_1, rs1015896_2, respectively.*

If you have not already done so, try the same logistic Lasso regression using the reduced dataset `gwas_small` instead. How does the tuning parameter λ change? How do estimated coefficients change? *A smaller tuning parameter* λ *is chosen by cross validation for the reduced dataset. The cross-validated model includes 19 SNPs. The four SNPs that were chosen by the full dataset are included but have larger coefficients. Indeed, generally speaking, the larger the model the larger the chosen value for* λ *is, and in relative terms (and many times in absolute terms too) cross-validated Lasso will choose more aggressive regularisation for larger dimension* p, *because we want to reduce the variance more in such high-dimensional models. The resulting coefficients are shrunk more towards zero with a higher level of regularisation.*

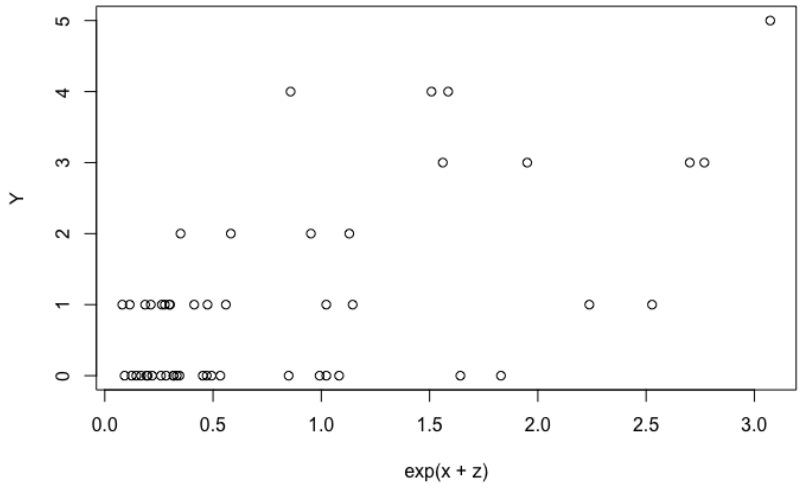
Practical 4: Overdispersion

To illustrate the consequences of overdispersion as a result of missing important covariates, we generate (or simulate) synthetic data from a Poisson model with two covariates, so

$$Y_i \overset{\text{ind}}{\sim} \text{Poi}(\mu_i), \qquad \log \mu_i = x_i + z_i,$$

for $i = 1, \ldots, 50$. We generate the covariates x_i, z_i as logs of independent gamma random variables:

```
> set.seed(2018)
> n <- 50
> x <- log(rgamma(n, 3, 2))
> z <- log(rgamma(n, 1, 2))
> Y <- rpois(n, lambda=exp(z+x))
```



Now let us fit the correct model, as well as an incorrect one which omits z_is.

```
> mod1 <- glm(Y ~ x+z, family=poisson)
> mod2 <- glm(Y ~ x, family=poisson)
```

The fact that we fail to measure z_i in the second model means that the responses will vary more than predicted by the model, i.e. there will be overdispersion. Recall from large sample and small dispersion asymptotics that the generalised Pearson statistic in both models should be approximately 1 and, if the model with p covariates is correct, its distribution should be $(n-p)^{-1}\chi^2_{n-p}$ approximately. Take a look at the following computations.

```
> phihat1 <- sum(residuals(mod1,type="pearson")^2)/mod1$df.residual
> phihat1
> 1-pchisq(phihat1*mod1$df.residual,mod1$df.residual)
> phihat2 <- sum(residuals(mod2,type="pearson")^2)/mod2$df.residual
> phihat2
> 1-pchisq(phihat2*mod2$df.residual,mod2$df.residual)
```

What do you observe? *Both estimates are around 1 with the first being a better approximation and the second being above 1 as expected. In the 5%-significance goodness-of-fit tests we cannot reject the first model, whilst we can reject the second.*

Get 95% confidence intervals for the regression parameters in each model, and compare to their true values.

```
> coef1 <- summary(mod1)$coefficients
> coef2 <- summary(mod2)$coefficients
> coef1[,1] + 1.96*cbind(-coef1[,2], coef1[,2])
> coef2[,1] + 1.96*cbind(-coef2[,2], coef2[,2])
```

What happens in the second case? *The confidence interval for the intercept fails to cover the true parameter value. The confidence interval for the x-covariate is narrower than in the first case. This is a result of the inferential procedures being overly optimistic in the second model, as a result of unaccounted variability. We check this below.*

Now let us check the coverage of our confidence intervals for, e.g., the x-covariate.

```
> N <- 1000
> cover2 <- cover1<-rep(0,N)
> for (i in 1:N) {
+     xs = log(rgamma(n, 3, 2))
+     zs = log(rgamma(n, 1, 2))
+     Ys = rpois(n, lambda=exp(zs+xs))
+
+     mod1bis = glm(Ys ~ xs+zs, family=poisson)
+     mod2bis = glm(Ys ~ xs, family=poisson)
+
+     coef1bis = summary(mod1bis)$coefficients
+     coef2bis = summary(mod2bis)$coefficients
+     CI1 = coef1bis[2,1] + 1.96*cbind(-coef1bis[2,2], coef1bis[2,2])
+     CI2 = coef2bis[2,1] + 1.96*cbind(-coef2bis[2,2], coef2bis[2,2])
+     cover1[i] <- (CI1[1] < 1) && (CI1[2] > 1)
+     cover2[i] <- (CI2[1] < 1) && (CI2[2] > 1)
+ }
> mean(cover1)
> mean(cover2)
```

What do we see? *The coverage of the first model is approximately correct, whilst that of the second is too small in line with the insights above.*

Self harm data

We now analyse overdispersion through seemingly zero-inflated real data. The data used here is from a randomised controlled trial to compare the effect of a brief form of cognitive therapy, manual assisted cognitive behaviour therapy (MACT), with treatment as usual (TAU) for recurrent self-harm. This data-set is only a subset of the full data-set and used here for illustrative purposes only.

Besides the number of self-harm episodes that took place over a 6-month period (`count`) and the treatment group (`trt`: TAU=0, MACT=1), information on the age (`age`), sex (`sex`: M = 0, F = 1), borderline personality disorder status (`bpd`: no personality disorder = 1, borderline personality disorder = 2, and other personality disorder = 3) and centre status (`centre`: Centre 1 = 0, Centre 2 = 1) is also provided. Read in the data into `R`.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "selfharm.csv"
> selfharm <- read.csv(paste0(filePath, fileName), header=T)
> head(selfharm)
> attach(selfharm)
```

Make sure you examine the data properly before proceeding. Produce relevant tables, plots and summary statistics for the data. Keep in mind what is the objective of the study. You may wish to use the `R` functions `table`, `summary`, `by`, `boxplot`. *For example, we can get descriptive information about the counts overall and by treatment group, and then produce box-plots of the counts by treatment group.*

```
> table(count)
> summary(count)
> sd(count)
> by(count, trt, summary)
> by(count, trt, var)

> boxplot(split(log(count),trt),names=c("TAU", "MACT"),
col=2:3, ylab= "Number of recurrent self-harm episodes")
```

We can also do cross-tabulations of `trt` with the various other binary/categorical variables using the R command `table`. E.g.,

```
> by(count,selfharm[,c("trt","sex")],table)
```

The following `R` libraries will be needed for the rest of the practical:

```
install.packages("MASS"); library(MASS)
install.packages("pscl"); library(pscl)
install.packages("lmtest"); library(lmtest)
```

We begin by producing histograms of the number of self harm episodes overall and by treatment group.

```
> truehist(count,h=1,col="grey")
> truehist(count[trt==0],h=1,col="red")
> truehist(count[trt==1],h=1,col="green")
```

What do you notice? *We notice the large proportion of zero counts.*

Let us first fit a Poisson model to the number of recurrent self-harm episodes.

```
> centre <- as.factor(centre)
> bpd <- as.factor(bpd)
> selfharm.pois <- glm(count~trt+age+centre+bpd*sex, family = "poisson")
> summary(selfharm.pois)
```

What do you observe? Is this expected? Can you interpret the results? Can you work out from the model fitted how many of the subjects would have been expected to not have any self-harm episodes during the 6-month period? You may wish to use the function `fitted`, which returns the fitted means for a given fitted model.

We observe that there is a strong indication of overdispersion because of the much larger residual deviance that residual degrees of freedom. This is somewhat expected since the histograms showed a larger proportion of zeroes, but we should set a statistical test to have a confidence statement of this kind.

If we were to ignore the overdispersion we would identify `trt`, `age`, `bpd3`, `sex` *and the* `bpd3:sex` *interaction effect as being statistically significant at 5%-level. In particular, we would interpret the treatment effect as providing evidence that MACT is effective at reducing the mean number of repeat episodes compared to TAU (rate ratio of 0.65 with 95%CI of (0.53, 0.79)).*

The expected number of subjects who would have no further episodes under this model can be calculated in `R` *as* `sum(exp(-fitted(selfharm.pois)))`*, which gives* 74.46*, and this can be compared to the observed number, 165, of zero counts. Make sure you understand why mathematically we used the above* `R` *command.*

Let us conduct a test using Pearson's Chi-squared statistic, i.e. the generalised Pearson statistic multiplied by the residual degrees of freedom.

```
> chisq <- sum(residuals(selfharm.pois,type="pearson")^2)
> chisq
> 1-pchisq(chisq,selfharm.pois$df.residual)
```

How do you interpret the result? *We note the large value (although we should divide it*

by the residual degrees of freedom to be able to compare it to something, in this case to 1). When testing it, we get that the p-value is highly significant and therefore we reject the null hypothesis of no extra-Poisson variation and conclude that there is strong evidence of overdispersion. This was to be expected from the heterogeneity in the large number of zero observations.

Negative binomial model. Now instead of fitting a Poisson regression model to the data we fit a negative binomial regression model. We use the `R` function `glm.nb` from library `MASS`:

```
> selfharm.nb <- glm.nb(count~trt+age+centre+bpd*sex)
> summary(selfharm.nb)
```

Note that the parametrisation on `R` uses $\theta = 1/\tau$ (so $\tau = 1/0.2095$ here), where τ is the overdispersion parameter.

Do you understand why this was done and do you understand the results? Can you work out how many subjects would have been expected to not have any recurrent self-harm episodes during the six-month period under this model? Does this model improve on the Poisson regression model fitted earlier? What is the overdispersion parameter? Have a look at the help file for `odTest` and then use it to compare the negative binomial to the Poisson.

As there is overdispersion, we have decided to begin by extending the Poisson model assuming that there is unobserved heterogeneity. This results in the mean-variance relationship $\operatorname{Var} Y_i = \mathbb{E}Y_i + \theta^{-1}(\mathbb{E}Y_i)^2$. *For this model there is no evidence of* `trt`, `age`, `bpd3`, `sex` *and the* `bpd3:sex` *interaction effects being statistically significant, unlike in the original Poisson model. To work out the expected number of subjects with zero counts we use the following R command*

```
> sum(dnbinom(0,mu=fitted(selfharm.nb),size=selfharm.nb$theta))
```

Again make sure you understand the mathematics behind this command. The overdispersion parameter is equal to $1/0.2095 = 4.77$. *From executing* `odTest(selfharm.nb)` *on* `R` *we infer that it is statistically significantly different from zero due to the p-value in the likelihood ratio test between the two models (null being Poisson) being* $< 2.2e-16$ *(you may also check this by hand with a bit of work). Therefore there is strong evidence against the Poisson model in favour of the negative binomial model.*

Zero-inflated models. Let us now attempt to explicitly account for excess zeroes by fitting various zero-inflated models. Under these models we assume that the excess zeroes are due to some subjects never again self-harming. We start by fitting zero-inflated

Poisson (ZIP) models using `zeroinfl` from library `pscl`:

```
> selfharm.zip1 <- zeroinfl(count~trt+age+centre+bpd*sex | 1, dist="poisson")
> selfharm.zip2 <- zeroinfl(count~trt+age+centre+bpd*sex, dist="poisson")
> summary(selfharm.zip1)
> summary(selfharm.zip2)
```

Make sure you understand what has been done here and can interpret the results! In the first model, the zero probability is the same for all Y_i, whereas in the second model, the zero probability follows a logistic model that depends on the same set of parameters as used in the Poisson model. What if you want to model zero probability in a logistic model that depends only on `trt` and `age` (and an intercept term)? Look at the R help file for `zeroinfl` if needed. *We modify the part of the formula after the bar*

```
> zeroinfl(count~trt+age+centre+bpd*sex | trt+age, dist="poisson")
```

We can also perform a likelihood ratio test to compare the two models fitted.

```
> lrtest(selfharm.zip1,selfharm.zip2)
```

What do you conclude? *There is not enough evidence (at 5% significance level) to reject the more parsimonious model `selfharm.zip1`, which assumes that the excess zero probability does not depend on covariates.*

Repeat the above exercise but now using zero-inflated negative binomial (ZINB) models instead of ZIP models:

```
> selfharm.zinb1 <- zeroinfl(count~trt+age+centre+bpd*sex | 1, dist="negbin")
> selfharm.zinb2 <- zeroinfl(count~trt+age+centre+bpd*sex, dist="negbin")
```

We can compare all proposed models using AIC (or BIC).

```
> models <- list(selfharm.pois, selfharm.nb, selfharm.zip1, selfharm.zip2,
> selfharm.zinb1, selfharm.zinb2)
> dim_models <- c(9, 10, 10, 18, 11, 19)
> AICs <- -2*sapply(models, logLik) + 2*dim_models # equivalently, we can use
# sapply(models, AIC)  to compute AIC; explicitly  computing the dimensions is
# more pedagogical
> BICs <- -2*sapply(models, logLik) + log(length(count))*dim_models
```

Make sure you understand how the dimensions were computed. Which model is selected by AIC/BIC? *The negative binomial model is selected using either criterion. Note that that its AIC and BIC values are not far from those of the zero-inflated negative binomial model with equal zero probability. If we believe that subpopulation is really "inactive" then we may still wish to choose the latter (we may also be missing some important covariate in the zero probability). Useful bonus: try `models[[which.min(AICs)]]` to*

extract an element from a list.

Practical 5: quasi-likelihood and generalised linear mixed models

Quasi-likelihood models

Synthetic data. Let us try fitting a quasi-Poisson model to the synthetic data from the beginning of the last practical to account for overdispersion.

```
> set.seed(2018)
> n <- 50
> x <- log(rgamma(n, 3, 2))
> z <- log(rgamma(n, 1, 2))
> Y <- rpois(n, lambda=exp(z+x))
> mod3 = glm(Y ~ x, family=quasipoisson)
> summary(mod3)
```

Why does AIC give NA? *Because to compute AIC we require a probability density function but in quasi-likelihood models like quasi-poisson we do not make distributional assumptions (only first and second moment assumptions).*

Compare the parameter estimates and standard errors from models `mod2` and `mod3`. *The parameter estimates are the same, but the standard errors are larger under `mod3`.*

Does the confidence interval cover the true parameter in `mod3`? *It does cover the true parameter (and regions are larger as they should be blown up to reflect the modelled overdispersion).*

Calculate the ratio of the standard errors for the slope parameter in models `mod3` and `mod2`. *The ratio is precisely the square root of the multiplier*[2]*, 1.257.*

Note that `R` uses a t-test rather than a z-test to perform hypothesis tests: can you see why? *Because we are estimating the multiplier, and $\check{\phi}/\phi$ has an approximate (scaled) χ^2 distribution. Note that both versions are asymptotically valid as $n \to \infty$.*

Now let us check the coverage of the quasi-Poisson confidence intervals.

[2]Typo: the numerical value is now correct, I probably used the wrong seed in the first version.

```
> N <- 1000
> cover3 <- rep(0,N)
> for (i in 1:N) {
+     xs = log(rgamma(n, 3, 2))
+     zs = log(rgamma(n, 1, 2))
+     Ys = rpois(n, lambda=exp(zs+xs))
+
+     mods = glm(Ys ~ xs, family=quasipoisson)  # fit the quasi-Poisson model
+     tmp = summary(mods)
+     betahat = tmp$coefficients[2,1]
+     se = tmp$coefficients[2,2]
+
+     CI3 = betahat + c(-1,1)*1.96*se
+     cover3[i] = (CI3[1] < 1) && (CI3[2] > 1)
+ }
```

What do you see? *The coverage of the quasi-Poisson models is approximately correct.*

Self harm data. We now move to the self harm data set. We calculate the multiplier estimate and use this to inflate the variances of our regression parameter estimates.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "selfharm.csv"
> selfharm <- read.csv(paste0(filePath, fileName), header=T)
> attach(selfharm)
> install.packages("MASS"); library(MASS)
> centre <- as.factor(centre)
> bpd <- as.factor(bpd)
> selfharm.pois <- glm(count~trt+age+centre+bpd*sex, family = "poisson")
> chisq <- sum(residuals(selfharm.pois,type="pearson")^2)
> phi <- chisq/selfharm.pois$df.residual
> summary(selfharm.pois,dispersion=phi)
```

What has been done here? How does this approach compare with fitting a `glm` with a quasi-Poisson family directly? *This was done to inflate the Poisson standard errors by a factor $\hat{\phi}^{1/2}$. The parameter estimates and standard errors are identical to fitting a quasi-Poisson directly:*

```
summary(glm(count~trt+age+centre+bpd*sex, family = "quasipoisson"))
```

However, as expected, R performs a t-test instead of z-test if specified with `family="quasipoisson"` directly.

Generalised linear mixed models

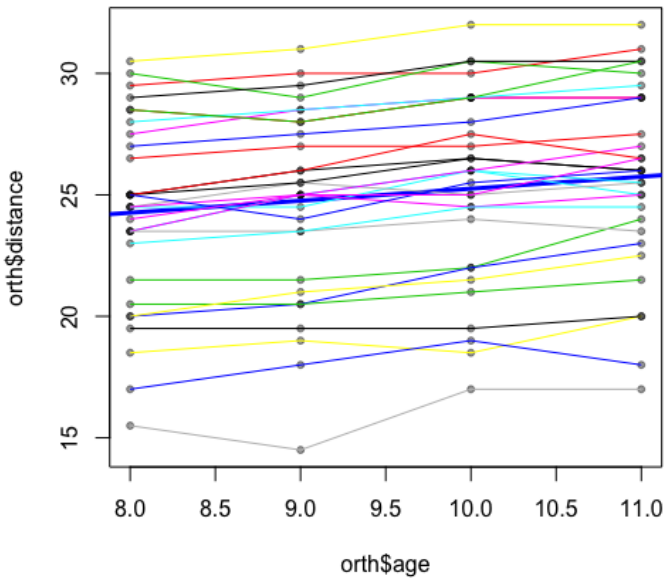
We now focus on how to fit generalised linear mixed effects models using the `lme4` package (we could alternatively use package `nlme`, both have very similar functionality).

```
> install.packages("lme4")
> library(lme4)
```

Orthodontic data. A dentist makes an orthodontic measurement on each of 30 children at ages 8, 9, 10 and 11. They are interested in modelling the growth in this measurement over time. We start by plotting the individual growth curves of all children studied in the dataset.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "Orthodont.txt"
> orth <- read.table(paste0(filePath, fileName), header = T)
> head(orth)

> # now plot data: 'alpha' sets transparency level to display overlapping points
>
> plot(orth$age, orth$distance, pch=20, col=rgb(0,0,0,alpha=0.4))
>
> # plot individual growth curves:
>
> for (i in 1:30) {
+     ind <- orth$pid==paste0("P",i)
+     points(orth$age[ind], orth$distance[ind],type="l", col=i)
+ }
```



What do you observe? *We observe that different individuals have different intercepts, and this difference is stochastically maintained as individuals age over the measurement period.*

For the ith individual's observation at time $j = 1, 2, 3, 4$, we will denote the quantity distance by Y_{ij}, and the individual's age by x_{ij}. Consider the linear model $Y_{ij} = \beta_0 + \beta_1 x_{ij} + \epsilon_{ij}$, where $\epsilon_{ij} \overset{\text{iid}}{\sim} N(0, \sigma^2)$. We already do not expect this model to be reasonable, as it ignores the different intercepts between individuals and the within group correlation between measurements. We fit it anyway for pedagogical reasons (and offset the age so that 8 years is the baseline).

```
> orth.lm1 <- lm(distance ~ I(age-8), data = orth)
> summary(orth.lm1)
> plot(orth.lm1, which = c(1,2,3,4))
```

Look at the diagnostic plots: what do you see? What is the moral of this? *There is nothing really to suggest any problems, which shows that diagnostic plots can't necessarily tell you a model is a bad idea! (Although note that the p-value based on the F-statistic in the summary is telling us to not reject the intercept-only model in favour of the fitted model...a sign that the proposed model is not reasonable.)*

If we wish to model the difference between individuals, we can of course try to incorporate `pid` as a covariate. However, inclusion of this categorical variable adds 29 additional parameters to the linear model. Moreover, if we are interested in estimating also the

effect of `sex`, inclusion of `pid` makes the model non-identifiable (the design matrix does not have full rank).

The crucial observation here is that though we need to acknowledge the dependence of `distance` on `pid` to make more accurate inference on the linear coefficient of `age`, we are not inherently interested in the exact parameter estimates of `pid` (i.e. the intercepts). The random effect model gives a more parsimonious description of the effect of `pid` by describing the intercepts as realisations from a normal distribution. In addition, it models the within-individual correlations too. Write down the algebraic form of the model, making sure you define each parameter and effect. *$Y_{ij} = \beta_0 + u_i + \beta_1 x_{ij} + \epsilon_{ij}$, where $u_i \overset{iid}{\sim} N(0, \tau^2)$, $\epsilon_{ij} \overset{iid}{\sim} N(0, \sigma^2)$ and u_is and ϵ_{ij}s are independent. Note that the u_is are (unmeasured) random variables, not parameters, and that the number of parameters does not grow as the number of individual increases.*

We can fit these models with the function `lmer` from `lme4`. Do so keeping in mind that the syntax is a little tricky. Start with

```
> orth.lme1 <- lmer(distance ~ I(age - 8) + (1 | pid), data=orth)
> summary(orth.lme1)
```

```
Linear mixed model fit by REML ['lmerMod']
Formula: distance ~ I(age - 8) + (1 | pid)
Data: orth

REML criterion at convergence: 347.7

Scaled residuals:
    Min      1Q  Median      3Q     Max
-2.4762 -0.5798 -0.0163  0.5046  1.9298

Random effects:
Groups   Name        Variance Std.Dev.
pid      (Intercept) 14.9512  3.867
Residual              0.2725  0.522
Number of obs: 120, groups:  pid, 30

Fixed effects:
            Estimate Std. Error t value
(Intercept) 24.26333    0.71044   34.15
I(age - 8)   0.49667    0.04262   11.65

Correlation of Fixed Effects:
           (Intr)
I(age - 8) -0.090
```

The term `(1|pid)` in the formula says that random effects should be at the level of intercepts (hence the `1`) for each individual. The summary gives a lot of output; check that you can identify (i) the estimates of β_0 and β_1; (ii) the estimates of σ and τ.

$\hat{\beta}_0 = 24.26 - 8 \times 0.497 = 20.32$, $\hat{\beta}_1 = 0.497$, $\hat{\sigma} = 0.522$, $\hat{\tau} = 3.867$.

Notice that the fit was done by REML by default. We can get a maximum likelihood fit by adding the option `REML=FALSE`. Fit the model with maximum likelihood, calling the object `orth.lme1b`. Compare your answers, noting what is different from `orth.lme1` and what is not. *The regression parameters are the same, but the variance parameter estimates are a bit smaller as expected.*

```
> orth.lme1b <- lmer(distance ~ I(age - 8) + (1 | pid), data=orth, REML=FALSE)
> summary(orth.lme1b)
```

Compare the estimated values and standard errors of τ^2, σ^2 in `orth.lme1b` to those in `orth.lme1`, what do you observe and is that expected? *Estimated values of τ^2 and σ^2 are both smaller in the maximum likelihood fit. The standard errors are also smaller in the maximum likelihood fit. Both phenomena are expected because the maximum likelihood estimates the variance parameters after fitting the fixed effect, which negatively biases the estimates. On the other hand, REML projects the data onto orthogonal complement of the column space of X to remove the fixed effect before estimating τ^2 and σ^2 and REML estimates are unbiased.*

Formally, the ordinary linear model and `orth.lme1b` are nested if we regard the former as the latter with $\tau^2 = 0$. Assume that the likelihood ratio test were valid. The function `logLik` returns log-likelihood values of linear models and mixed effects models.

```
>lrstat <- 2*(logLik(orth.lme1b)[1] - logLik(orth.lm1)[1])
>lrstat
```

What do you conclude? *The LR test statistic is 318.9, though it is not appropriate to directly compare with a χ^2_1-distribution (see below), such a large value highly suggests that the random effects term is important.*

How valid is the assumption that the likelihood ratio statistic is approximately chi-square distributed? Let us assume that the linear model fitted in `orth.lme1b` with no random effect is the true model and generate an empirical distribution of the likelihood ratio test statistic (we do all under MLE rather than REML for a coherent comparison).

```
> set.seed(2018)
> B <- 100; lrstatboots <- rep(0,B)
> attach(orth)
> for (b in 1:B){
+     distanceboots <- 24.26 + (age - 8) * 0.497 + rnorm(30*4, sd=0.5191)
+     # could also have chosen sd=3.869; under the null, this should not matter
+
+     boot.lm <- lm(distanceboots ~ I(age - 8))
+     boot.lme <- lmer(distanceboots ~ I(age - 8) + (1 | pid), REML=FALSE)
+     #  returns singularity warnings as data comes from the null, so no random effects
+
+     lrstatboots[b] <- 2*(logLik(boot.lme)[1] - logLik(boot.lm)[1])
+ }
> lrstatboots
> sum(lrstatboots<1e-10)
```

What do you observe? *Just over half of the test statistics are zero (or very near zero). So the chi-square approximation is not valid.*

The synthetic data `distanceboots` generated in the above manner is known as *parametric bootstrap samples*. If the null model is true, the MLE parameters in the linear mixed effects model should be close to those of the linear model, and for large `B` the empirical distribution in `lrstatboots` is close to the likelihood ratio test statistic under the null hypothesis. We can therefore construct a parametric bootstrap p-value of the null hypothesis by comparing `lrstat` to the vector `lrstatboots`.

```
> boot.pval <- sum(lrstatboots>lrstat)/B
> boot.pval
```

In this case, since the likelihood ratio statistic is so large, the bootstrap p-value is also essentially zero and we reject the null in favour of the alternative.

Next let us fit a model with both a random intercept and a random slope.

```
orth.lme2 <- lmer(distance ~ I(age - 8) + (I(age - 8) | pid), data=orth)
summary(orth.lme2)
```

Write down the algebraic form of the model being fitted and identify the coefficient estimates.

$$Y_{ij} = \beta_0 + u_{i0} + (\beta_1 + u_{i1})x_{ij} + \epsilon_{ij}$$

where $\epsilon_{ij} \overset{iid}{\sim} N(0, \sigma^2)$,

$$u_i = (u_{i0}, u_{i1})^\top \overset{iid}{\sim} N\left(0, \begin{pmatrix} \tau_0^2 & \rho\tau_0\tau_1 \\ \rho\tau_0\tau_1 & \tau_1^2 \end{pmatrix}\right)$$

and $\{\epsilon_{ij} : i, j\} \perp\!\!\!\perp \{u_i : i\}$. *Coefficient estimates*[3]*:* $(\hat{\beta}_0, \hat{\beta}_1) = (24.26 - 8 \times 0.497, 0.497) =$

[3]Typo: changed numerical value of $\hat{\beta}_0$, as x_{ij} is the age rather than `age` $- 8$.

$(20.32, 0.497)$, $(\hat{\tau}_0, \hat{\tau}_1, \hat{\rho}) = (3.904, 0.024, -1)$ *and* $\hat{\sigma} = 0.521$.

Construct a test to verify if the additional random effect in the slope included in `orth.lme2` significant (warning: `orth.lme1` and `orth.lme2` nested or only formally?). *The two are only formally nested. Thus, we conduct a parametric bootstrap likelihood ratio test between the two models:*

```
> lrstat <- 2*( logLik(orth.lme2)[1] - logLik(orth.lme1)[1])
> set.seed(2018)
> B <- 100; lrstatboots <- rep(0,B)
> for (b in 1:B){
+     distanceboots <- 24.26 + (age - 8) * 0.497 + rep(rnorm(30, sd=3.904), each=4)
                       + rnorm(30*4, sd=0.521)
+     boot.lme1 <- lmer(distanceboots ~ I(age - 8) + (1 | pid))
+     boot.lme2 <- lmer(distanceboots ~ I(age - 8) + (I(age - 8) | pid))
+     lrstatboots[b] <- 2*(logLik(boot.lme2)[1] - logLik(boot.lme1)[1])
+ }
> lrstatboots
> boot.pval <- sum(lrstatboots>lrstat)/B
> boot.pval
> ## 0.83
```

The p-value is 0.83. So we do not reject the simpler random-intercept model.

Word frequency data

We look at the `writtenVariationLijk` dataset in `languageR` package:

```
install.packages("languageR")
library(languageR)
```

This dataset records the number of times words ending in -lijk appear in Dutch newspapers and was used in a socio-geographical study. It contains the following variables:

- `Corpus`: a factor with levels the sampled newspapers: `belang` (Het Belang van Limburg), `gazet` (De Gazet van Antwerpen), `laatnieu` (Het Laatste Nieuws), `limburg` (De Limburger), `nrc` (NRC Handelsblad), `stand` (De Standaard), and `tele` (De Telegraaf).
- `Word`: a factor with the 80 most frequent words ending in -lijk.
- `Count`: a numeric vector with token counts.
- `Country`: a factor with levels `Flanders` and `Netherlands`.
- `Register`: a factor with levels `National`, `Quality` and `Regional` coding the type of newspaper.

We are interested in knowing how word use varies across country and newspapers. The word frequency certainly also depends on the word itself, but we are uninterested in the exact relation and would prefer to treat the by-word variation as a random effect.

```
> data(writtenVariationLijk)
> head(writtenVariationLijk)
> lijk.glme1 <- glmer(Count ~ Country + Register + (1|Word),
+                     data = writtenVariationLijk, family = "poisson")
> summary(lijk.glme1)
```

To call the `glmer` function above, you may need to install and load the `nlme` library:

```
install.packages("nlme")
library(nlme)
```

(even if it does not seem to install correctly, try to run `glmer` anyway). The `glmer` function fits a generalised linear mixed effect model by maximum likelihood (there is not yet a satisfactory way to generalise REML to generalised linear mixed effect models, so a REML fit is not implemented in `glmer`). How to interpret the estimated random effect coefficient? *The logarithm of mean count per word has a standard deviation of 0.854 across all words with suffix '-lijk'.*

How to interpret the estimated coefficient for `CountryNetherlands`? *For each word with suffix '-lijk', the mean word usage count in a newspaper in Netherlands is* $\exp(0.165)$ *times that in a newspaper in Flanders, controlling for the rest of covariates.*

Why do you think there is a table of correlations for the fixed effects? *It shows the correlations between the estimators of the different coefficients. This is because we use the glmm to account for correlations and the table shows the resulting ones after fitting the model. You may compare it with the output of*

```
> lijk.glm <- glm(Count ~ Country + Register , data = writtenVariationLijk,
                  family = "poisson")
> cov2cor(vcov(lijk.glm))
```

Let us perform some tests now.

```
> lijk.glme2 <- glmer(Count ~ Country * Register + (1|Word),
+                     data = writtenVariationLijk, family = "poisson")
> lijk.glme3 <- glmer(Count ~ Country * Register + (Country|Word),
+                     data = writtenVariationLijk, family = "poisson")
> anova(lijk.glme1, lijk.glme2)
> anova(lijk.glme2, lijk.glme3)
```

The `anova` function compares two models using their likelihood ratio test statistic. Are the two tests valid? *The first test is valid since the additional parameter is in the fixed*

effect, and the null parameter of 0 is in the interior of the parameter space. The second test is not valid since we are testing whether separate random effects are needed for the two countries. The null parameter is hence on the boundary of the parameter space and the distributional limit of the likelihood ratio test statistic is not a chi-squared distribution. We should use a parametric bootstrap test in this case. However, we expect that this limiting distribution is a mixture of a Dirac delta at zero and a chi-squared distribution on one degree of freedom so, the parametric bootstrap test statistic is typically smaller than the chi-square null distribution and since we observe an extremely large likelihood ratio test statistic, we would expect the parametric bootstrap test to produce a p-value very near 0 as well and reject the null hypothesis anyway.

Practical 6: Linear discriminant analysis and logistic classifiers

This practical looks at two different methods of fitting linear classifiers. The linear discriminant analysis is implemented in the `MASS` package and the logistic classifier is implemented in the `nnet` package:

```
library(MASS)
install.packages("nnet")
library(nnet)
```

Iris data

The `iris` dataset is available in the `MASS` package. This is the classical dataset Fisher used in his original 1936 paper on linear discriminant analysis. The dataset contains observations of iris flowers. Each observation consists of measurements of the mean length and width of both the flower sepals and petals, followed by the species of the flower. Read in the data and perform some exploratory analysis.

```
> data(iris)  # load data
> head(iris)
> pairs(iris[,1:4])
```

LDA. Now, perform linear discriminant analysis using the `lda` function from the `MASS` package and interpret the output.

```
> iris.lda <- lda(Species~., data=iris)
> iris.lda

# Call:
#   lda(Species ~ ., data = iris)
#
# Prior probabilities of groups:
#   setosa versicolor  virginica
# 0.3333333  0.3333333  0.3333333
#
# Group means:
#            Sepal.Length Sepal.Width Petal.Length Petal.Width
# setosa            5.006       3.428        1.462       0.246
# versicolor        5.936       2.770        4.260       1.326
# virginica         6.588       2.974        5.552       2.026
#
# Coefficients of linear discriminants:
#                     LD1         LD2
# Sepal.Length  0.8293776  0.02410215
# Sepal.Width   1.5344731  2.16452123
# Petal.Length -2.2012117 -0.93192121
# Petal.Width  -2.8104603  2.83918785
#
# Proportion of trace:
#    LD1    LD2
# 0.9912 0.0088
```

The 'prior probabilities' are the estimated group priors $\hat{\pi}(\ell)$ and the 'group means' are the class centroids $\hat{\mu}_\ell$, $\ell = 1, 2, 3$. For the rest of the output we need to introduce the concept of *sphering*, which in turn gives another interpretation of LDA as a data dimension-reduction tool.

Sphering refers to the transformation of the data $\tilde{x} := \hat{\Sigma}^{-1/2}x$. If $\hat{\Sigma} \to^P \Sigma$ and $\hat{\mu}_\ell \to^P \mu_\ell$ as $n \to 0$ for all $\ell \in \mathcal{L}$, then $\tilde{x}|_{Y=\ell} \to^d N(\Sigma^{-1/2}\mu_\ell, I_p)$ as $n \to \infty$, i.e., the asymptotic contours of the density of the transformed data (conditioned on $Y = \ell$) are spheres rather than ellipsoids, hence the name to the transformation. Let $\tilde{\mu}_\ell := \hat{\Sigma}^{-1/2}\hat{\mu}_\ell$ be the sphered class centroids, let $\mathcal{C}$ be the linear subspace of $\mathbb{R}^p$ parallel to the affine space they span $\{\sum_{\ell=1}^L \lambda_\ell \tilde{\mu}_\ell : \sum_{\ell=1}^L \lambda_\ell = 1\} = \{\tilde{\mu}_L + \sum_{\ell=1}^{L-1} \lambda_\ell(\tilde{\mu}_\ell - \tilde{\mu}_L) : \lambda_\ell \in \mathbb{R}, \ell = 1, \ldots, L-1\}$ (which has dimension $\leq \min\{p, L-1\}$), and let $\Pi_\mathcal{C}$ be the projection onto it. Then, the last term on the right hand side of

$$\tilde{x} - \tilde{\mu}_\ell = \Pi_\mathcal{C}(\tilde{x} - \tilde{\mu}_\ell) + (I_p - \Pi_\mathcal{C})\tilde{x} - (I_p - \Pi_\mathcal{C})\tilde{\mu}_\ell$$

does not depend on ℓ and, since LDA for a given x chooses the class that maximises

$$-\frac{1}{2}(x - \hat{\mu}_\ell)^T\hat{\Sigma}^{-1}(x - \hat{\mu}_\ell) + \log\hat{\pi}(\ell) = -\frac{1}{2}\|\tilde{x} - \tilde{\mu}_\ell\|_2^2 + \log\hat{\pi}(\ell),$$

we can use Pythagoras theorem to conclude that LDA maximises

$$-\frac{1}{2}\|\Pi_\mathcal{C}(\tilde{x} - \tilde{\mu}_\ell)\|_2^2 + \log\hat{\pi}(\ell).$$

In words, LDA projects the (sphered) data onto $\mathcal{C}$ (reducing the dimension of the data considerably if $\dim(\mathcal{C}) << p$) and, there, it chooses the closest sphered centroid after adjusting for class proportions[4].

Let $\tilde{B} \in \mathbb{R}^{p\times p}$ be the covariance matrix of $\tilde{C} := (\tilde{\mu}_1, \ldots, \tilde{\mu}_L)^\top$ accounting for group sizes, i.e. if $N_\ell := \sum_{y_i=\ell} 1$ and $\bar{\tilde{\mu}} := n^{-1}\sum_{\ell=1} N_\ell \tilde{\mu}_\ell = n^{-1}\sum_{i=1}^n \tilde{x}_i$ then[5]

$$\tilde{B} = \frac{1}{L-1}\sum_{\ell=1}^{L} N_\ell(\tilde{\mu}_\ell - \bar{\tilde{\mu}})(\tilde{\mu}_\ell - \bar{\tilde{\mu}})^\top,$$

and let $\tilde{B} = \tilde{V}\tilde{D}\tilde{V}^\top$ be its eigen-decomposition with the entries of $\tilde{D}$ ordered decreasingly. Then, the $r = \operatorname{rank}(\tilde{B}) \leq \min\{p, L-1\}$ linear discriminants are

$$V = \hat{\Sigma}^{-1/2}\left(\tilde{V}_{\cdot 1}, \ldots, \tilde{V}_{\cdot r}\right).$$

In particular, the matrix of 'coefficients of linear discriminants' in the output above is

$$V = \hat{\Sigma}^{-1/2}\left(\tilde{V}_{\cdot 1}, \tilde{V}_{\cdot 2}\right) = \begin{pmatrix} 0.8293776 & 0.02410215 \\ 1.5344731 & 2.16452123 \\ -2.2012117 & -0.93192121 \\ -2.8104603 & 2.83918785 \end{pmatrix},$$

from which we can construct $\Pi_{\mathcal{C}} = \hat{\Sigma}^{-1/2} V V^\top \hat{\Sigma}^{-1/2}$. The 'proportion of trace' are

$$\frac{1}{\sum_{s=1}^r \tilde{D}_{ss}}\left(\tilde{D}_{11}, \ldots, \tilde{D}_{rr}\right)$$

(known as the Rayleigh quotients due to an alternative interpretation of LDA whose details go beyond the scope of the course). In this example they are equal to 0.9912 and 0.0088, so most of the signal is captured in a single discriminant direction. Note that V herein only includes the first two columns of $\tilde{V}$ because the dimension of $\mathcal{C}$ is 2. If $r > 2$, we may still project the (sphered) data onto the $K < r$ discriminant directions capturing the most of the signal, which is called reduced rank LDA (and, usually, $K = 2$ for visualisation purposes).

We can visualise the LDA output using the following `plot` command. See `?plot.lda` for more details.

```
> plot(iris.lda)
```

How are the x- and y- coordinates of the two-dimensional plot computed? *The two-dimensional plot is obtained by first sphering the data points and then projecting them*

[4] Typo: added clarifications to the next paragraph introducing $r = \operatorname{rank}(\tilde{B})$ and corrected the computation of the 'proportion of trace'.

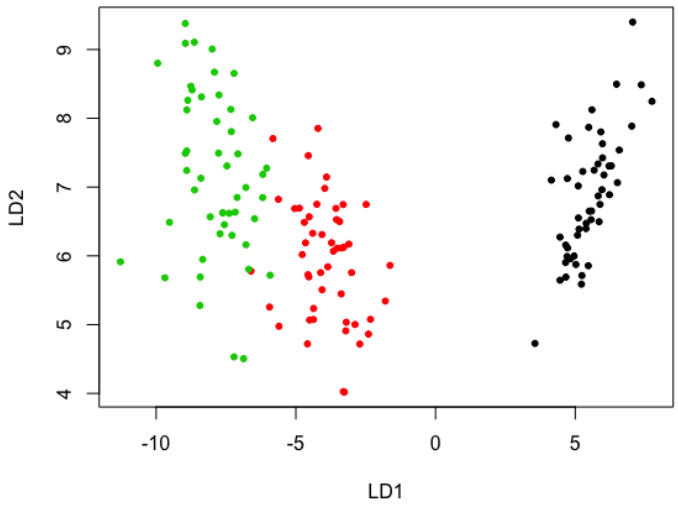
[5] Typo: $\tilde{\mu}_L <-> \tilde{\mu}_\ell$ below

onto the affine subspace spanned by all three class centroids after sphering. Mathematically, if X is the data matrix (with each row representing one observation), the plotted data here are rows of XV after centring them, where V is the "coefficient of linear discriminant" matrix above given in the `R` output.

We can manually obtain the same plot (and colour coding the classes) in the following two ways.

```
> V <- iris.lda$scaling
> X <- as.matrix(iris[,1:4])
> plot(X%*%V, pch=20, col=iris$Species)

> pred <- predict(iris.lda, newdata=iris) # another way to obtain projected points
> plot(pred$x, pch=20,col=iris$Species) # predict also centres the data
```



What is the training misclassification error of LDA? What is the leave-one-out cross-validated misclassification error? *Using the `predict` function, we obtain that the training error is 2% and the cross-validated error is also 2%.*

```
> sum(pred$class != iris$Species)/150  # training err
> cvfit <- NULL
> for (i in 1:150){
+ cvfit[i] <- predict(lda(Species~., data=iris[-i,]), newdata=iris[i,])$class
+ }
sum(cvfit != as.numeric(iris$Species))/150  # cross-validated err
```

Logistic classifier. Let us now apply logistic regression classification on the same dataset. This can be done using the `multinom` function in the `nnet` package (this will

silently fit a (shallow) neural network!). Of course, if it had been a two-class classification problem, we could have used the good old `glm` function too.

```
> iris.logit <- multinom(Species~., data=iris)
> coef <- t(coef(iris.logit))
> coef
#              versicolor  virginica
# (Intercept)   18.408209 -24.230061
# Sepal.Length  -6.082250  -8.547304
# Sepal.Width   -9.396625 -16.077164
# Petal.Length  16.170374  25.599633
# Petal.Width   -2.058115  16.227474
```

For which covariate values will a flower be classified as `virginica`? *It is described by the following inequality (remember the coefficients for the baseline class `setosa` are all 0)*

$$-24.23 - 8.55\texttt{Sepal.length} - 16.08\texttt{Sepal.width} + 25.6\texttt{Petal.length} + 16.23\texttt{Petal.width} > \max\{-18.41 - 6.08\texttt{Sepal.length} - 9.40\texttt{Sepal.width} + 16.17\texttt{Petal.length} - 2.06\texttt{Petal.width}, 0\}.$$

The misclassification training error using logistic classifier is 1.3%.

```
> sum(predict(iris.logit, newdata=iris) != iris$Species)/150
# 0.013333333
```

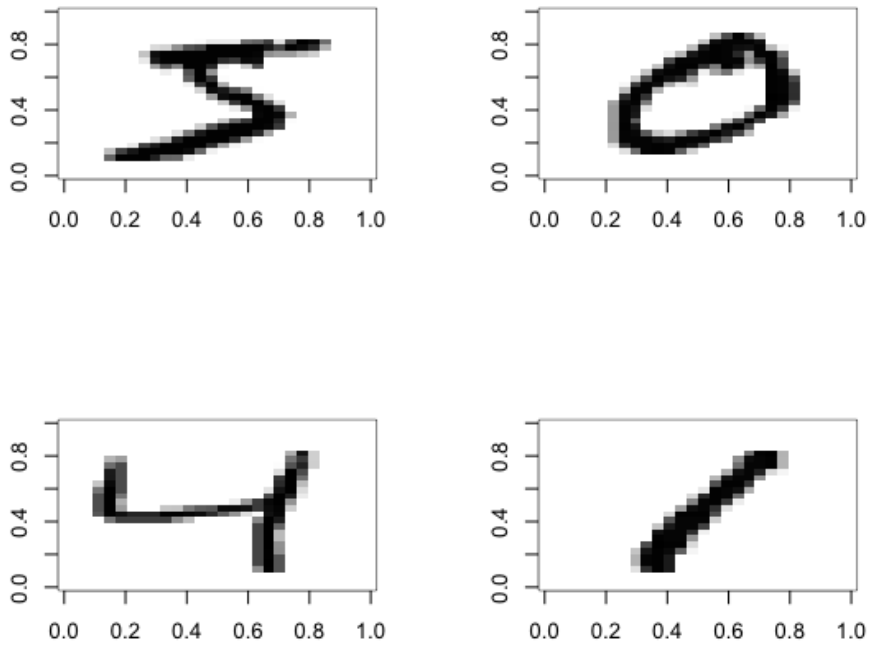
MNIST data

The MNIST (Modified National Institute of Standards and Technology) data is a database of handwritten digits. We will be using a subset of the database.

The full database can be found on `http://yann.lecun.com/exdb/mnist/`. Load the data into `R` and explore a bit.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "mnist.csv"
> mnist <- read.csv(paste0(filePath, fileName), header = TRUE)
> mnist[1:10,1:10]
> mnist$digit <- as.factor(mnist$digit)
> visualise = function(vec, ...){ # function for graphically displaying a digit
+     image(matrix(as.numeric(vec),nrow=28)[,28:1], col=gray((255:0)/255), ...)
+ }
> old_par <- par(mfrow=c(2,2))
> for (i in 1:4) visualise(mnist[i,-1])
> par(old_par)
```

Each handwritten digit is stored in a 28×28 pixels grayscale image. The grayscale values (0 to 255) for the 784 pixels are stored as row vectors in the dataset. We define a `visualise` function to graphically display a digit based on its vector of grayscale values.



We use the first 2/3 of the data as training data and the remaining 1/3 as test data. Moreover, many margin pixels are constantly white throughout the dataset. We exclude them from our analysis.

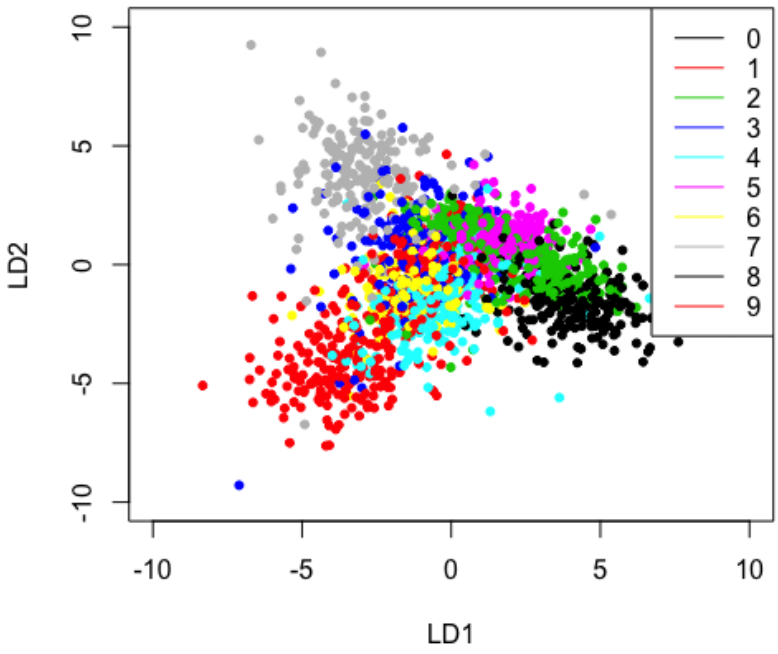
```
> train <- mnist[1:4000,]
> identical <- apply(train, 2, function(v){all(v==v[1])})
> # 2 indicates that the function will be applied over the columns
> train <- train[,!identical]
> test <- mnist[4001:6000,!identical]
```

LDA. We first fit a LDA classifier to the data. The test error is 16.9%.

```
> mnist.lda <- lda(digit~., data=train)
> pred <- predict(mnist.lda, test)
> sum(pred$class != test$digit)/2000
```

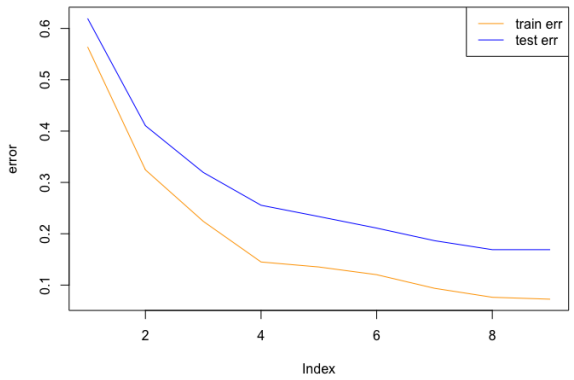
We can visualise the outcome by plotting along the first two canonical directions.

```
> plot(pred$x, pch=20, col=as.numeric(test$digit)+1, xlim=c(-10,10), ylim=c(-10,10))
> legend("topright", lty=1, col=1:10, legend=0:9)
```



The first two canonical directions already show relatively good separation of the 10 classes. The `dim` argument in `predict.lda` can be used to explicitly fit a reduced rank LDA classifier. We plot how the training error and test error change as more dimensions are included.

```
> train.err <- test.err <- rep(0, 9)
> for (r in 1:9){
+     train.err[r] <- sum(predict(mnist.lda, train, dim=r)$class != train$digit)/4000
+     test.err[r] <- sum(predict(mnist.lda, test, dim=r)$class != test$digit)/2000
+ }
> plot(train.err,type="l", col="orange",
+      ylim=range(c(train.err,test.err)), ylab="error")
> points(test.err,type="l", col="blue")
> legend("topright", c("train err", "test err"), col=c("orange", "blue"), lty=1)
```



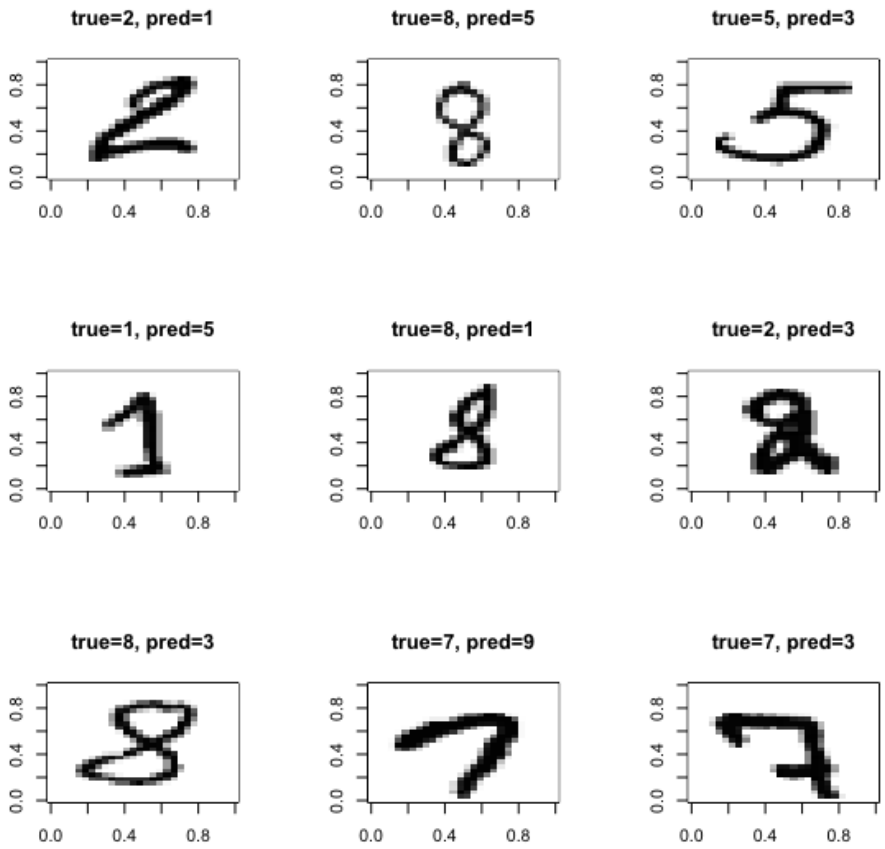
We see that all useful information is essentially captured in the first four to six canonical directions.

Logistic classifier. Next, we try to fit a logistic classifier to the MNIST data. We again use the `multinom` function. The `MaxNWts` argument controls maximum number of coefficients in the multinomial logistic model. The test error is 22.2%.

```
> mnist.logit <- multinom(digit~., data=train, MaxNWts=10000)
> sum(predict(mnist.logit, newdata=test) != test$digit)/2000
```

Let us have a look some of the misclassified images (non-examinable code).

```
pred <- (0:9)[max.col(cbind(0,x.test%*%beta))]
err_ind <- (1:2000)[pred != test[,1]]
old_par <- par(mfrow = c(3,3))
for (i in 1:9){
visualise(mnist[4000+err_ind[i],-1],
main=paste0("true=", test[err_ind[i],1], ", pred=", pred[err_ind[i]]))
}
par(old_par)
```


true=2, pred=1
true=8, pred=5
true=5, pred=3
true=1, pred=5
true=8, pred=1
true=2, pred=3
true=8, pred=3
true=7, pred=9
true=7, pred=3
0.0
0.4
0.8

Practical 7: Support vector machines

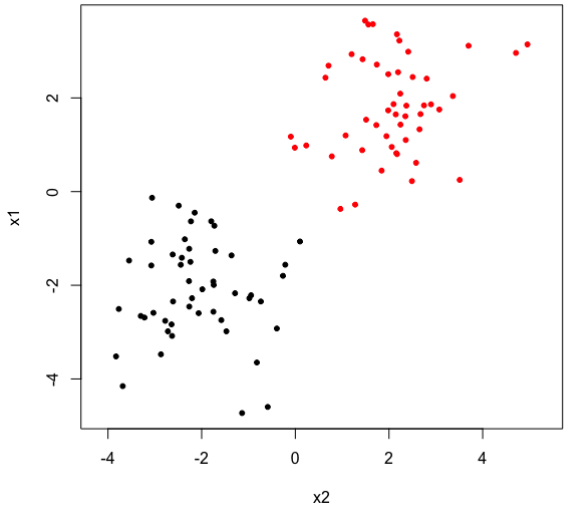
Support vector machines are implemented in several `R` packages, including `e1071`, `caret` and `kernlab`. We will use the `e1071` package in this practical:

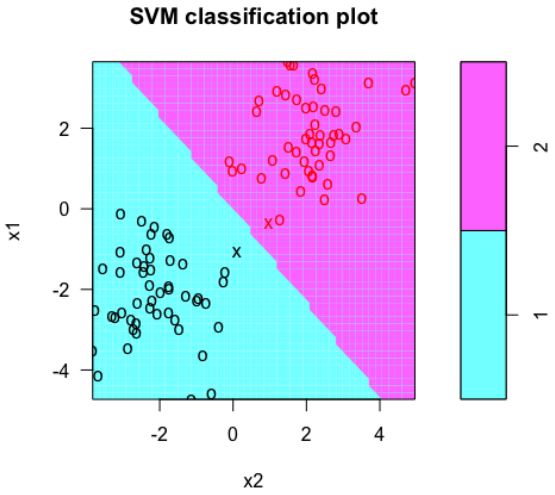
```
install.packages("e1071")
library(e1071)
```

An illustrative example

We first illustrate the ideas behind SVM using a simple simulated dataset. We suppose that the true distributions for the two classes are bivariate Gaussian with means $(-2, -2)^\top$ and $(2, 2)^\top$, respectively, and identity covariance matrices. Multivariate Gaussians can be simulated using the `mvrnorm` function in `MASS`.

```
> library(MASS)
> n <- 100
> set.seed(2018)
> x <- rbind(mvrnorm(n/2, c(-2,-2), diag(2)), mvrnorm(n/2, c(2,2), diag(2)))
> y <- as.factor(rep(c(1,2), each = n/2))  # class labels
> dat <- data.frame(x1 = x[,1], x2 = x[,2], y)
> with(dat, plot(x2, x1, pch=20, col=y, asp = 1))
```





We see that these two classes are linearly separable. The SVM classifier, shown in the

figure on the right and computed as follows, chooses the linear decision boundary that best separates the two classes.

```
> svm1 <- svm(y~x1+x2, data = dat, kernel = "linear", cost = 1000)
```

Ignore the argument `kernel = "linear"` for now. To understand argument `cost` we need to introduce an alternative formulation of SVM. Recall that, given $N \geq 0$, the optimisation problem in SVM is

$$\max_{\substack{\beta_0 \in \mathbb{R}, \\ \beta \in \mathbb{R}^p : \|\beta\|_2 = 1}} M \qquad \text{subject to } \; y_i(\beta_0 + x_i^\top \beta) \geq M(1 - \xi_i), \; \sum_{i=1}^n \xi_i \leq N, \; \xi_i \geq 0, \; i = 1, \ldots, n,$$

where the condition $\|\beta\|_2 = 1$ guarantees uniqueness. The value 1 is arbitrary, so we may take it to be $\|\beta\|_2 = 1/M$ and rephrase the problem as

$$\max_{\beta_0 \in \mathbb{R}, \beta \in \mathbb{R}^p} \frac{1}{\|\beta\|_2} \qquad \text{subject to } \; y_i(\beta_0 + x_i^\top \beta) \geq 1 - \xi_i, \; \sum_{i=1}^n \xi_i \leq N, \; \xi_i \geq 0, \; i = 1, \ldots, n.$$

Equivalently, we can minimise $\|\beta\|_2$, or the more computationally-friendly quantity $\|\beta\|_2^2/2$. Lastly, we can proceed as in ridge or lasso regression and write condition $\sum_{i=1}^n \xi_i \leq N$ as a regularisation term in the objective function, arriving at the alternative SVM optimisation problem

$$\min_{\beta_0 \in \mathbb{R}, \beta \in \mathbb{R}^p} \frac{1}{2}\|\beta\|_2^2 + C \sum_{i=1}^n \xi_i \qquad \text{subject to } \; y_i(\beta_0 + x_i^\top \beta) \geq 1 - \xi_i, \; \xi_i \geq 0, \; i = 1, \ldots, n.$$

Then, the argument `cost` is used to specify the regularisation or cost parameter C, which is in one-to-one correspondence with N. The function `svm` does not allow us to fit a hard margin directly. If the data is linearly separable, we can achieve a hard margin by setting a large C as we did here, as it will aggressively penalise the slack variables.

Recall that the support vectors are the data points that satisfy the constraint with equality rather than with strict inequality; thus, they determine the optimal hyperplane. Examine the output of `svm1`.

```
> summary(svm1)
> x[svm1$index,] # coordinates of the support vectors
> plot(svm1, dat)  # see ?plot.svm for more info
```

How many support vectors are there? Can you give an explicit description of the classification boundary based on the support vectors in this case? *There are two support vectors (marked with a cross in the plot). The linear classification boundary is simply the perpendicular bisector of the two support vectors.*

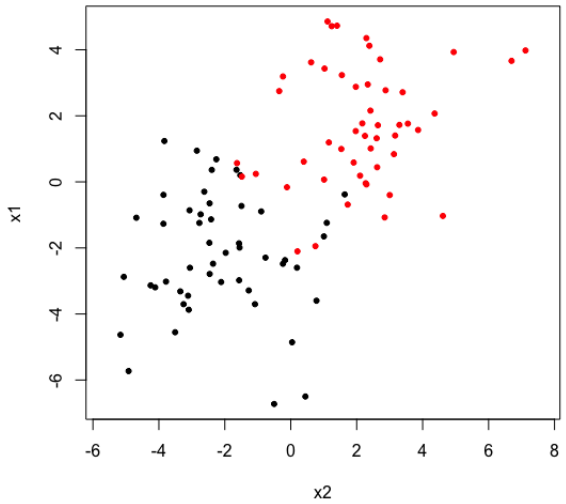
Let us see how this performs if we have some new data from the same distributions.

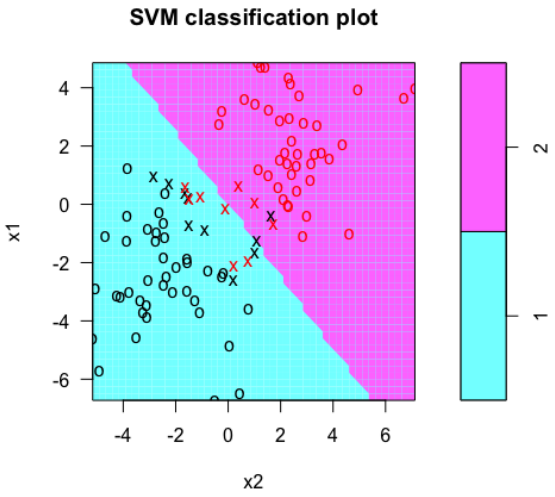
```
> xnew <- rbind(mvrnorm(n/2, c(-2,-2), diag(2)), mvrnorm(n/2, c(2,2), diag(2)))
> ynew <- rep(c(1,2), each = n/2)
> ypred <- predict(svm1, xnew)
> table(ypred, ynew)  # check predicted labels
> sum(ynew != ypred)/n  # prediction error
> # 0.01
```

The out-of-sample prediction error is 1% in this case.

Now we increase the variance in the two classes, so that no separating hyperplane exists. Let us fit SVM with cost parameter $C = 1$ and see again how it performs for new data.

```
> set.seed(2018)
> x <- rbind(mvrnorm(n/2, c(-2,-2), 3*diag(2)), mvrnorm(n/2, c(2,2), 3*diag(2)))
> y <- as.factor(rep(c(1,2), each = n/2))  # class labels
> dat <- data.frame(x1 = x[,1], x2 = x[,2], y)
> with(dat, plot(x1, x2, pch=20, col=y, asp = 1))
> svm2 <- svm(y~x1+x2, data = dat, kernel = "linear", cost = 1)
> summary(svm2)
> plot(svm2, dat)
> xnew <- rbind(mvrnorm(n/2, c(-2,-2), 3*diag(2)), mvrnorm(n/2, c(2,2), 3*diag(2)))
> ynew <- rep(c(1,2), each = n/2)
> ypred <- predict(svm2, xnew)
> sum(ynew != ypred)/n  # prediction error
> # 0.06
```





How many support vectors are there? What is the interpretation behind the last formulation of SVM? *19 support vectors. The SVM trades off the two objectives of fitting a*

hyperplane that has a large margin to the majority of the data points and minimising the total cost of data points falling on the wrong side of the margin.

Change the `cost` argument to `0.1`, `10`, `100`. How does the number of support vectors change? What happens if `cost` goes to infinity? *As `cost` increases, we impose a heavier penalty for misclassification and we have fewer support vectors in the optimal solution. Some will remain regardless of the value of C as the data is not perfectly separated. Indeed, as `cost` goes to infinity, the problem essentially becomes finding a hyperplane with smallest number of misclassifications (weighted by distance from the misclassified points to the optimal hyperplane).*

Kernel SVM

We can also handle the overlapping classes in the previous dataset by using non-linear kernels. The argument `kernel` in the `svm` function allows us to use some of the most popular ones:

- d^{th} degree polynomial kernel, $K(x, x') = (\gamma_0 + \gamma x^\top x')^d, \gamma_0, \gamma \in \mathbb{R}, d \in \mathbb{N}$;
- radial basis or Gaussian kernel, $K(x, x') = \exp(-\gamma \|x - x'\|_2^2), \gamma \in \mathbb{R}$; and,
- sigmoid or neural network kernel, $K(x, x') = \tanh(\gamma_0 + \gamma x^\top x'), \gamma_0, \gamma \in \mathbb{R}$.

The linear kernel, as specified in the previous section, corresponds to $K(x, x') = x^\top x'$, i.e. to not kernelising. Choosing a good kernel depends on the problem at hand and it is not a trivial task. Indeed, a careful understanding of it is beyond the scope of this course. It is generally chosen so that it embeds some desirable properties, such as $K(x, x')$ being large for nearby data points x, x' (so K is a measure of similarity), $K(x, x')$ being invariant under a certain data transformation T (e.g. rotation or translation, so $K(Tx, Tx') = K(x, x')$), or that the underlying feature map ϕ has certain regularity properties (so K determines regularity properties of the decision boundaries); we may choose its parameters (or the kernel itself if a few options are sensible) to optimise the prediction error by, e.g., cross validation. The three kernels above are standard as a result of consistently delivering good results and encompassing some of these properties; we note in passing that the third is not a positive definite kernel.

Let us fit a Gaussian kernel to the last data. In this case, there are two tuning parameters involved in the Gaussian kernel SVM: C and γ. Let us get some intuition on the role of these. Repeat the following lines of code for a few values of `cost` and γ (fixing one and varying the other).

```
> svm3 <- svm(y~x1+x2, data=dat, kernel = "radial", gamma = 1, cost = 1000)
> plot(svm3, dat)
> ypred <- predict(svm3, xnew)
> sum(ynew != ypred)/n  # prediction error
> # 0.11
```

What do you observe? *For fixed γ we see the role of C much more clearly than in the non-kernelised case: the larger it is, the stronger the penalty for misclassification and, since perfect separation is likely to have occurred in the new feature space, it leads to overfitting in the original feature space. The role of γ is similar, but for a different reason: the larger it is the more irregularities the kernel can model so the more it will overfit. In essence, we clearly see a bias-variance trade-off for both parameters.*

A common way to search for optimal parameters is to use a "line search", or "grid search" when there are multiple parameters to be tuned. We search through a geometrically spaced grid of parameter values and evaluate them via cross-validation. The tune function in the e1071 package can be used for this purpose (it performs 10-fold cross-validation by default).

```
> tune.out <- tune(svm, y~x1+x2, data=dat, kernel="radial",
> ranges = list(cost=10^c((-3):3), gamma=10^c((-3):3)))
> summary(tune.out)
```

This picks $C = 1$ and $\gamma = 0.01$, which give cross-validation error of 7%. Run it again. What values does it pick? What are the out-of sample prediction errors for these two combinations of parameters? *The second run gives $C = 10$ and $\gamma = 1$. We appropriately modify the penultimate block of code to obtain 6% and 9%.*

MNIST data

We apply the SVM classifier to the MNIST data that we have seen in the last practical. By default, SVM is a two-class classification method. A popular way to use it for L-class classification is by performing pairwise two-class classifications and classifing a data point according to the most commonly seen label in the $L(L-1)/2$ pairwise comparisons. Indeed, this is what the `svm` function does.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "mnist.csv"
> mnist <- read.csv(paste0(filePath, fileName), header = TRUE)
> mnist$digit <- as.factor(mnist$digit)
>
> train <- mnist[1:4000,]
> identical <- apply(train, 2, function(v){all(v==v[1])})
> train <- train[,!identical]  # remove redundant pixels
> test <- mnist[4001:6000,!identical]
>
> mnist.svm1 <- svm(digit~., data = train, kernel = "linear")  # this can be slow
> pred <- predict(mnist.svm1, test)
> table(test$digit, pred)  # confusion matrix
> sum(test$digit != pred) / 2000  # test error
> # 0.095
>
> mnist.svm2 <- svm(digit~., data = train, kernel = "radial")  # this can be slow
> pred <- predict(mnist.svm2, test)
> table(test$digit, pred)  # confusion matrix
> sum(test$digit != pred) / 2000  # test error
> # 0.093
```

By default, the `svm` function uses $C = 1$. The first model above therefore corresponds to soft margin SVM, whilst the second corresponds to radial kernel SVM with default value $\gamma = 1$. In practice, we typically search through a list of possible tuning parameters and choose the best model via cross-validation. However, we will not do it here since the cross-validation will be too slow, but we can of course improve upon the linear kernel more to a test error of around 7%. For comparison, the test errors of LDA and LRC were 16.9% and 22.2%, respectively.

Practical 8: Neural networks

Properly building and training a neural network involves many design decisions such as choosing number and nature of layers and fine-tuning hyperparameters. Many details are beyond the scope of this course. Single hidden-layer fully-connected neural networks can be implemented using the `nnet` package in `R` installed and loaded above. But serious network construction and training are usually carried out using the `keras` package. In this practical, we will see a few examples of using `keras` to build and train neural networks.

The `keras` package is a frontend `R` interface that runs the `TensorFlow` library from `python` below the hood. To install it follow the instructions below. **NB: Windows and Mac OSs may have issues, Linux is recommended for this practical; you may download it and set up a dual boot or use the computers in the CATAM room GL.04.**

```
install.packages("devtools")
devtools::install_github("rstudio/keras") # update all if prompted
library(keras)
install_keras() # install Miniconda if prompted
```

Lever data

Let us start with some synthetic data. We simulate the physical state of a lever (tilting to the left or right) given some left weight (`lw`) placed at a certain distance (`ld`) from the pivot and some right weight (`rw`) placed at a certain distance (`rd`) from the pivot.

```
> set.seed(2018)
> n <- 10000; ntrain <- 6000;
> lw <- runif(n); rw <- runif(n)
> ld <- runif(n); rd <- runif(n)
> tilt <- as.factor(sign(rw*rd-lw*ld))
> levels(tilt) <- c("L","R")  # change to more meaningful class labels
> dat <- data.frame(tilt,lw,rw,ld,rd)
> head(dat)
#   bal         lw         rw        ld        rd
# 1   R 0.33615347 0.75721924 0.5214453 0.8626043
# 2   L 0.46372327 0.03319081 0.3524001 0.8706108
# 3   R 0.06058539 0.59687077 0.7590107 0.6443922
# 4   L 0.19743361 0.24328849 0.6682254 0.3350507
# 5   L 0.47431419 0.43350584 0.4704572 0.0141134
# 6   R 0.30104860 0.50187735 0.2418792 0.3174299
```

Given that the state or label depends on the magnitude of products `lw*ld` and `rw*rd`, predicting the tilting is a fairly difficult problem for a neural network with the usual activation functions.

We use the first `ntrain` data points for training and the remaining for testing.

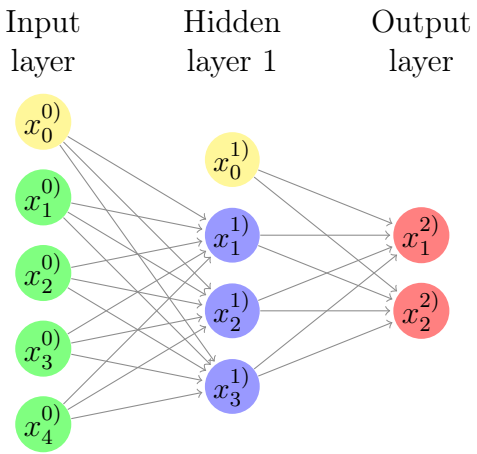
```
> x_train <- as.matrix(dat[1:ntrain,-1])
> x_test <- as.matrix(dat[-(1:ntrain),-1])
> y_train <- model.matrix(~tilt-1, data=dat[1:ntrain, ])
> y_test <- model.matrix(~tilt-1, data=dat[-(1:ntrain), ])
```

What is the effect of the final two lines above? *They convert the output label from a categorical variable to a binary indicator vector* $(\mathbb{1}_{\texttt{tilt="L"}}, \mathbb{1}_{\texttt{tilt="R"}})^\top$.

Let us first fit a fully-connected feedforward neural network model with one hidden layer of 10 nodes (without counting bias), using sigmoid activation and softmax output rule. We need to describe our architecture to `R`. This is achieved in the `keras` library via the following lines.

```
> hidden_layer <- layer_dense(units = 10, activation = "sigmoid", input_shape = c(4))
> output_layer <- layer_dense(units = 2, activation = "softmax")
> model <- keras_model_sequential(list(hidden_layer, output_layer))
> summary(model)
```

The `keras_model_sequential` function in `keras` builds a feedforward network by specifying layers in sequence in a list. Note that the input layer is not separately described but specified as `input_shape` in the first hidden layer. The `layer_dense` function builds a layer that is fully connected to the previous layer. Can you draw a schematic diagram for the neural network described above when the hidden layer has 3 nodes instead? *See figure below.*



How many neurons are there in the model of the code above? $5+11+2 = 18$. *Don't forget the bias neurons.* How many parameters are there in the model? Do you understand where that number comes from? *From summary, we see there are 72 parameters. This comes from* $(4+1) \times 10 + (10+1) \times 2$.

We then prepare the network for training by specifying the loss function and optimiser. Herein, the `categorical_crossentropy` loss coincides with the negative log-likelihood. The `metrics="acc"` argument allows us to monitor the training progress via training accuracy (1 minus the training error). We use stochastic gradient descent here.

```
> compile(model, optimizer="sgd", loss="categorical_crossentropy", metrics="acc")
```

The `compile` function does not generate an output but rather alter the `model` argument directly. In other words, the optimiser, loss and metrics are incorporated into `model` after the `compile` function. The actual training is carried out using the `fit` function: the weights are updated directly in `model` after the `fit` function.

```
> fit(model, x_train, y_train, epochs=20, batch_size=10)
```

Minibatch stochastic gradient descent is used here. It computes the stochastic gradient from `batch_size=10` observations at a time. When `batch_size` is equal to 1, we have the vanilla stochastic gradient descent and when `batch_size` is equal to `ntrain`, we recover the gradient descent. After fitting the model, we can test it on the test dataset using the `evaluate` function.

```
> evaluate(model, x_test, y_test)
```

The final training accuracy at the end of 20th epoch is 92.5%, and the test accuracy is 91% (you may get slightly different numbers due to the randomisation before each epoch in SGD, which is performed externally in Python).

Let us see if we can improve the model fit by adding more layers. Create two more hidden layers with 10 nodes each and build a 5-layer neural network. We then train this neural network as before using mini-batch stochastic gradient descent.

```
> hidden_layer <- layer_dense(units = 10, activation = "sigmoid", input_shape = c(4))
> hidden_layer2 <- layer_dense(units = 10, activation = "sigmoid")
> hidden_layer3 <- layer_dense(units = 10, activation = "sigmoid")
> output_layer <- layer_dense(units = 2, activation = "softmax")
> model <- keras_model_sequential(list(hidden_layer, hidden_layer2,
+                                      hidden_layer3, output_layer))
> wt0 <- get_weights(model) # store the initialised weights for later reference
> compile(model, optimizer="sgd", loss="categorical_crossentropy", metrics="acc")
> fit(model, x_train, y_train, batch_size=10, epochs=20)
> evaluate(model, x_test, y_test)
```

The final training accuracy is 52% and test accuracy is 49%. This 'deep' neural net is not doing much better than random guessing or flipping a coin! This may be surprising at first but we can find the reason for it (and fix it) by looking at how the weights are updated, i.e. by understanding how the gradient is computed. For this, we need to take

a slight detour and introduce the technique of *backpropagation*: due to the composition structure of feedforward neural networks, we can use the chain rule to compute the gradient iteratively and with local information only; you should check that, for a loss function $L(y, x), x, y \in \mathbb{R}^L$, and under the notation in lectures,

$$\frac{\partial R_i}{\partial \beta_{jk}^{h)}} = \delta_{ij}^{h)} x_k^{h-1)}, \quad i = 1, \ldots, n,\ j = 1, \ldots, p_h,\ k = 0, \ldots, p_{h-1},\ h = 1, \ldots, H+1,$$

where

$$\delta_{ij}^{h)} := \frac{\partial R_i}{\partial s_j^{h)}} = g_h' \left(s_j^{h)} \right) \sum_{m=1}^{p_{h+1}} \delta_{im}^{h+1)} \beta_{mj}^{h+1)}, \quad i = 1, \ldots, n,\ j = 1, \ldots, p_h,\ h = 1, \ldots, H,$$

and

$$\delta_{ij}^{H+1)} := n \sum_{\ell=1}^{L} \frac{\partial L \left(y_i, \underline{x}^{H+1)} \right)}{\partial x_\ell^{H+1)}} \frac{\partial x_\ell^{H+1)}}{\partial s_j^{H+1)}}, \quad i = 1, \ldots, n,\ j = 1, \ldots, L;$$

then, the gradient of the loss function for a feedforward neural network can be evaluated via a two-pass algorithm: (1) given a parameter $\hat{\theta}$ and a vector of covariates x_i, compute $\hat{\underline{x}}^{h)}$ in order $h = 1, \ldots, H+1$ (forward pass); (2) given $\hat{\theta}$, the output of (1) and a response y_i, compute $\left(\hat{\delta}_{ij}^{h)} \right)_{j,h}$ and $\left(\nabla_{\underline{\beta}_j^{h)}} R_i(\hat{\theta}) \right)_{j,h}$ in reverse order $h = H+1, \ldots, 1$ (backward pass).

We can get considerable intuition about some potential pitfalls in the training of feedforward neural networks simply by looking at the expressions above. In particular, note that $g_h' \left(s_j^{h)} \right)$ features in $\partial R_i / \partial \beta_{jk}^{h)}$ multiplicatively. Then, the following are some common pitfalls:

- *Saturation*: if $g_h'(s) \approx 0$ for certain values of $s \in \mathbb{R}$—e.g., $|s|$ large for the sigmoid activation function or $s < 0$ for ReLU—and $s_j^{h)}$ is in such range, $\beta_{jk}^{h)}$ will not change much in a (stochastic) gradient descent an update; in such case we say that node $x_{ik}^{h)}$ is *saturated* and, if ReLU is used, such node may stay saturated forever which is called a *dead ReLU unit*.

- *Symmetry breaking*: to avoid saturation at the beginning of training, we generally standarise input data, e.g., to have mean 0 and variance 1, and initial weights are chosen to be small; but note that if we choose all parameters to be exactly zero and $g_h(0) = 0$—as for the ReLU activation function—, $\underline{\beta}_j^{h)}$ will not change in a (stochastic) gradient descent update (except perhaps the bias weights for the output layer), and if $g_h(0) \neq 0$—as for the sigmoid activation function—, $\underline{\beta}_j^{h)}$ will change but will be equal within each layer (except perhaps for the first hidden layer) thus making the model useless; hence, we break the symmetry by taking $\beta_{jk}^{h)} \overset{\text{iid}}{\sim} \text{Unif}(-c, c)$ with $c = 0.7$ typically.

- *Vanishing gradients*: for the sigmoid function, $g'(s) \leq 1/4$ for all $s \in \mathbb{R}$; therefore, if the sigmoid is used in all layers, gradients with respect to weights in layers away from the output one may be very small even if no saturation is present.

The third pitfall explain the recent popularity of ReLU as opposed to the classical sigmoid function—which was introduced as a smooth version of the step function that modelled the activation of a neuron from enough signal.

We can now return to understanding the changes in the weights of the last fitted network, noting that the `compile` function includes the backpropagation rules and implements symmetry breaking at the start of training.The `get_weights` function in the code above above is used to extract weights. Weights between successive layers are stored as matrices of the form $(\beta_{uv})_{u,v}$ (for a non-bias node v in the previous layer and a non-bias node u in the next layer) and vectors of the form $(\beta_{u0})_u$ (weight from the bias node in the previous layer to a non-bias node u in the next layer). Let us reinitialise the neural network with stored weights in `wt0` and run the model for one epoch and see how the weights change.

```
> set_weights(model, wt0)
> fit(model, x_train, y_train, batch_size=10, epochs=1)
> wt1 <- get_weights(model)
> wt1[[1]]-wt0[[1]] # compare weights of the first hidden layer
```

We see that the weights did not move much after a whole training epoch. Due to the (unstandarised) inputs having similar units and reasonable sizes, at the beginning of training we do not expect saturation. The most likely culprit for the failure to training is vanishing gradient. As anticipated, we can rectify this by using a rectifier activation instead.

```
> hidden_layer <- layer_dense(units = 10, activation = "relu", input_shape = c(4))
> hidden_layer2 <- layer_dense(units = 10, activation = "relu")
> hidden_layer3 <- layer_dense(units = 10, activation = "relu")
> output_layer <- layer_dense(units = 2, activation = "softmax")
> model <- keras_model_sequential(list(hidden_layer, hidden_layer2,
+                                      hidden_layer3, output_layer))
> wt0 <- get_weights(model) # store the initialised weights for later reference
> compile(model, optimizer="sgd", loss="categorical_crossentropy", metrics="acc")
> fit(model, x_train, y_train, batch_size=10, epochs=20)
> evaluate(model, x_test, y_test)
```

We get a final epoch training accuracy of 97% and test accuracy of 95%. You can again check how much the weight moves after one epoch using the same commands as before. The movements are much more substantial.

MNIST data

Let us look at the MNIST data again (remember that this is a subset of around 10% of the actual MNIST data).

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "mnist.csv"
> mnist <- read.csv(paste0(filePath, fileName), header = TRUE)
> x_train <- as.matrix(mnist[1:4000,-1])
> y_train <- mnist[1:4000,1]
> x_test <- as.matrix(mnist[4001:6000,-1])
> y_test <- mnist[4001:6000,1]
```

```
> x_train <- x_train / 255
> x_test <- x_test / 255
> y_train <- as.factor(y_train)
> y_test <- as.factor(y_test)
> y_train <- model.matrix(~y_train-1)
> y_test <- model.matrix(~y_test-1)
```

We first try a fully-connected single-hidden-layer feedforward network.

```
> hidden_layer <- layer_dense(units = 256, activation = "relu", input_shape = c(784))
> output_layer <- layer_dense(units = 10, activation = "softmax")
> model <- keras_model_sequential(list(hidden_layer, output_layer))
> summary(model)
> wt0 <- get_weights(model)
> str(wt0)
```

What are the architecture (just description), activation rule and output rule of the neural network fitted? *Fully connected single-hidden-layer feedforward network with rectifier activation and softmax output rule. Not counting bias nodes, there are 784 input layer nodes, 256 hidden layer nodes and 10 output nodes.*

As before, we prepare the network for training by specifying a loss function and optimiser, then carry out the actual training using the `fit` function. After fitting the model, we can test it on the test dataset using the `evaluate` function.

```
> compile(model, optimizer="sgd", loss="categorical_crossentropy", > metrics="acc")
> fit(model, x_train, y_train, epochs=10, batch_size=10)
> evaluate(model, x_test, y_test)
```

The test accuracy is about 91%. How many parameters are there in the model? *There are* $(784+1)\times 256+(256+1)\times 10=203530$ *parameters.* How is this lack of over-fitting explained? *The ratio between the number of parameters and the training sample size is* 51 *(!) and we obtained a test accuracy of* 91% *without using any penalisation procedures.*

This lack of over-fitting may seem surprising at first: lurking in the background is that we run SGD for only 10 epochs or, in other words, that we applied early stopping in SGD; this acts as a form of regularisation to prevent over-fitting. Indeed, it is customary to choose the number of epochs so that the testing error is minimised. Remember that the loss function of a neural network will have (possibly infinitely) many minima, and we are not interested in finding a global minimum unlike with MLE as this would lead to over-fitting.

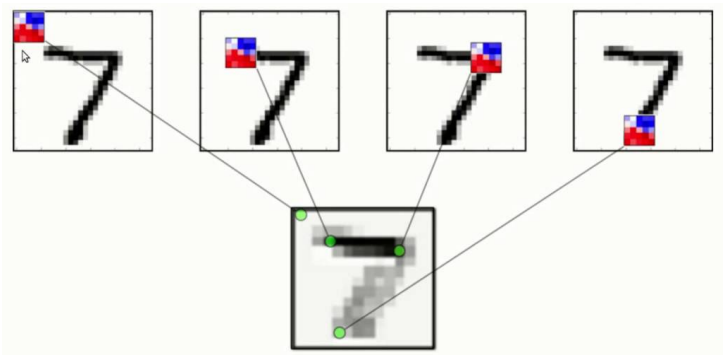
By default, SGD is run with a constant learning rate, which does not satisfy a slow decay assumption for the theoretical guarantee mentioned in the course notes (specifically, it requires $\sum_{e=1}^{\infty} \alpha_e^2 < \infty$ and $\sum_{e=1}^{\infty} \alpha_e = \infty$). We do not cover how to change this, but it can be done by giving a different input to the argument `optimizer` when calling the `compile` function (cf. https://keras.io/optimizers/).

Given the set up, we do not expect saturation or vanishing gradients. Nonetheless, let us see how the weights have changed. The following code compares the weights of the hidden layer before and after training.

```
wt1 <- get_weights(model)
wt1[[1]] - wt0[[1]]
```

It seems that the difference is 0 in many places (but check that they are not the same using e.g. `sum(wt1[[1]] != wt0[[1]])`). We see many 0s because the first few rows of the weight matrix have not moved at all during training. Why are these weights not updated? *Since the first few pixels are constantly white in all input images, the corresponding units satisfy $x_j^{0)} = 0$. By the comments in the symmetry breaking pitfall above, these components of the gradient are always zero, and the corresponding parameters never move in the SGD updates after initialisation. This is less worrying than saturation or vanishing gradient (after all, the test accuracy is not bad), but it suggests that a more careful initialisation of the weights values may lead to better results.*

(Optional/<u>non-examinable</u>) In tasks such as image recognition, it makes sense to construct networks with local connectivity and shared weights. More precisely, each hidden unit is only connected to a small patch of nodes in the previous layer. In the example of image analysis, these small patches correspond to fixed-shape localities in the picture. The same weights are used to combine pixels in each locality. The shared weights therefore extracts the same features in different parts of the image. In such networks, rather than taking linear combinations of the units, the units are convolved with compactly supported kernels of weights. Hence they are called *convolutional neural networks*.

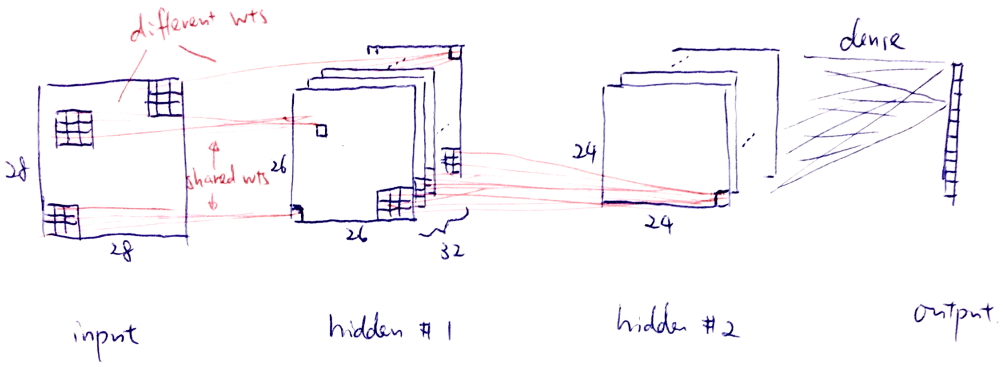


We can fit such and more complicated neural networks following the same principles as above. Let us fit a two-hidden layer *convolutional neural network*. The `array_reshape` function can be used to retain the geometry of the image (instead of flattening it to a vector).

```
x_train <- array_reshape(x_train, dim=c(4000,28,28,1))
x_test <- array_reshape(x_test, dim=c(2000,28,28,1))
architecture <- list(
layer_conv_2d(filters=32, kernel_size=c(3,3), activation = "relu",
input_shape = c(28,28,1)),
layer_dropout(rate=0.4),
layer_conv_2d(filters=32, kernel_size=c(3,3), activation = "relu"),
layer_dropout(rate=0.4),
layer_flatten(),
layer_dense(units = 10, activation = "softmax")
)
model <- keras_model_sequential(architecture)\\
summary(model)
```

Note that we have used a popular regularisation method called *dropout*: for each data point and before each gradient evaluation, it removes a hidden and an input node (from the original net; i.e. we train a subgraph of the original) with probabilities p (typically ≈ 0.5) and q (typically ≈ 0.2), respectively, so that a hidden node cannot rely on others to fix its mistakes; and, for testing, we use all nodes with hidden weights multiplied by p, so that the resulting model is some sort of average of all "dropped-out nets". Dropout decreases computational cost, the learning rate can be increased in comparison to SGD and it can be combined with other regularisation methods.

We train this neural net with a dropout rate $p = 0.4$ for each of the two hidden layers and $q = 0$ for the input layer. This is achieved via a 'dropout layer'. The two convolutional layers each have a kernel window of size 3×3 and are each measuring 32 features. The `layer_flatten()` removes the geometry and flattens all nodes into a vector. See below for a sketch the architecture of the convolutional neural network. Bias nodes are not shown.



We note that the first convolutional layer has $32 \times 26 \times 26 \times (9+1) = 216320$ connections to the input layer, but only $32 \times (9+1) = 320$ parameters, since the weights are shared across nodes for each feature in a convolutional layer. Similarly, each node in the second convolutional layer depends on the 32×9 nodes in the previous layer (the 3×3 window is applied to all 32 features), together with the bias, we have $(32 \times 9 + 1) \times 32 = 9248$ parameters. The output layer is fully connected to the previous convolutional layer and has $(32 \times 24 \times 24 + 1) \times 10 = 184300$ parameters.

We use a different optimiser 'adadelta' this time (it's a variant of stochastic gradient descent where we use larger step sizes for weights that haven't seen much movement in the past few updates). This more complicated architecture gives an accuracy of about 95.3%, which is quite high given our small training dataset.

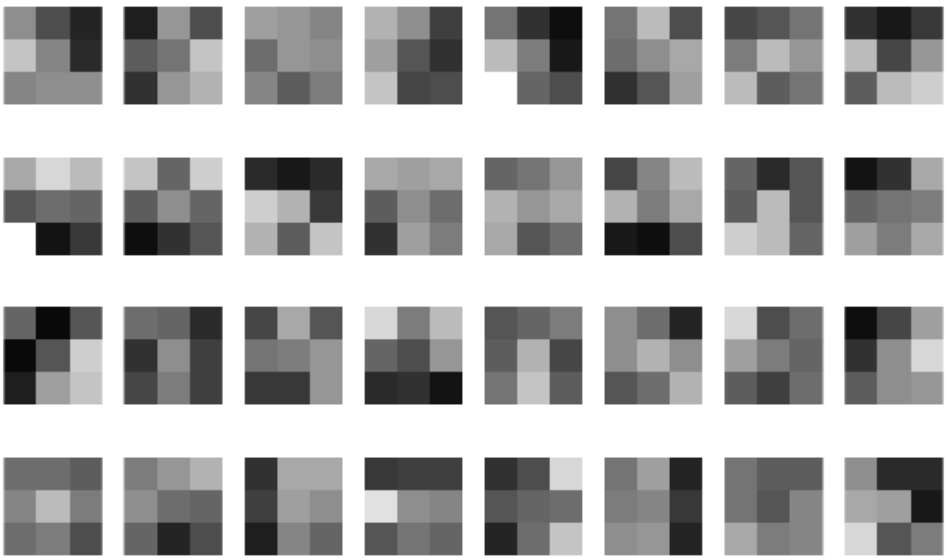
```
compile(model, loss="categorical_crossentropy", optimizer="adam", metrics="acc")
fit(model, x_train, y_train, epochs = 5, batch_size = 10)
evaluate(model, x_test, y_test)
```

We can also extract the weight matrix for convolutional layers.

```
wt <- get_weights(model)
str(wt)
# List of 6
# $ : num [1:3, 1:3, 1, 1:32] 0.098 0.0602 -0.0537 -0.022 -0.0988 ...
# $ : num [1:32(1d)] -0.02787 -0.00178 -0.00938 -0.00284 -0.02815 ...
# $ : num [1:3, 1:3, 1:32, 1:32] 0.0277 -0.022 -0.035 -0.1171 -0.0422 ...
# $ : num [1:32(1d)] -0.02183 0.00502 -0.02203 0.01087 -0.0114 ...
# $ : num [1:18432, 1:10] 0.00264 -0.00213 -0.01456 -0.00589 0.01557 ...
# $ : num [1:10(1d)] -0.040366 -0.002503 -0.008397 -0.018596 -0.000744 ...
```

The weight matrix for the first convolutional layer is really a three-dimensional array in this case. Each of the 32 features has a 3×3 convolutional filter. We can visualise these convolutional filters as follows.

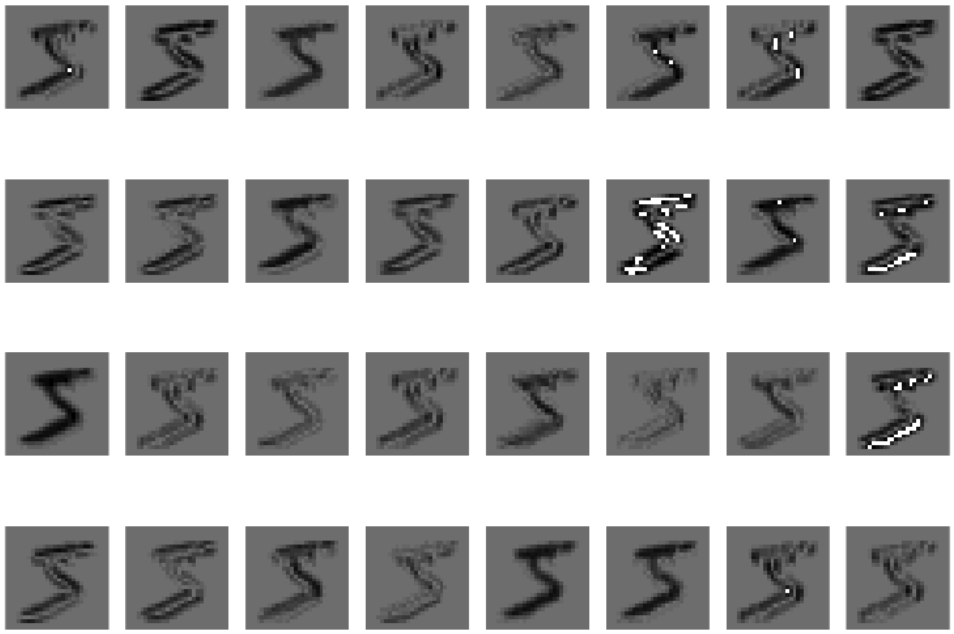
```
old_par <- par()
par(mfrow = c(4,8), mar=c(0.5,0.5,0.5,0.5))
for (i in 1:32){
mx <- wt[[1]][,,1,i]
image(t(mx[3:1, ]), asp=1,axes=F,frame.plot=F,zlim=range(-0.3,0.3),
col=gray((0:32)/32))
}
par(old_par)
```



These three-by-three grid of grayscale images represent values of the filter (black means negative values, white positive values). Let us see what these filters do to a specific image.

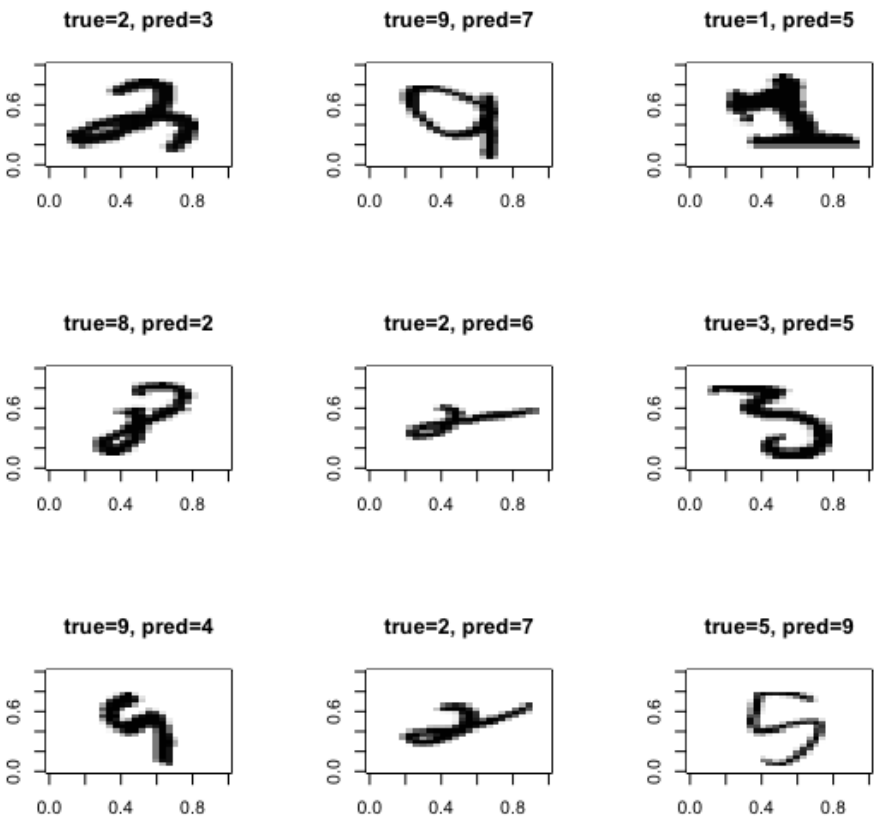
```
img = x_train[1,,,1]
par(mfrow = c(4,8), mar=c(0.5,0.5,0.5,0.5))
for (r in 1:32){
convout = matrix(0,28,28)
for (i in 2:27) for (j in 2:27){
convout[i,j] = sum(img[(i-1):(i+1), (j-1):(j+1)] * wt[[1]][,,1,r])
}
image(abs(t(convout[28:1,1:28])), axes=F,asp=1,col=gray((32:0)/32))
}
par(old_par)
```

From the output image, we can deduce that some of the convolutional filters are extracting straight edges along certain directions. For example, the 24th filter is extracting straight lines roughly along ENE-WSW direction, corresponding to the bottom stroke of the figure 5 here.



As before, we explore which of the handwriting digits were misclassified. These are definitely more 'interesting' mistakes than ones from the LDA or logistic classifiers.

```
err_ind <- (1:2000)[pred != mnist[4001:6000,1]]
visualise = function(vec, ...){ # function for graphically displaying a digit
image(matrix(as.numeric(vec),nrow=28)[,28:1], col=gray((255:0)/255), ...)
}
old_par <- par(mfrow = c(3,3))
for (i in 1:9){
visualise(mnist[4000+err_ind[i],-1],
main=paste0("true=", y_test[err_ind[i]], ", pred=", pred[err_ind[i]]))
}
par(old_par)
```


true=2, pred=3
true=9, pred=7
true=1, pred=5
true=8, pred=2
true=2, pred=6
true=3, pred=5
true=9, pred=4
true=2, pred=7
true=5, pred=9
0.0
0.4
0.8
0.6

Practical 9: Nearest neighbour classifiers

Nearest neighbour classifiers and variants are implemented in the `FNN` package:

```
install.packages("FNN")
library(FNN)
```

Synthetic data

We start by classifying the two-dimensional Gaussian-mixture dataset from the `ElemStatLearn` package, which is no longer available for new versions of `R`. You may still find it on the website of the book:

```
> website <- "https://web.stanford.edu/~hastie/ElemStatLearn/datasets/ESL.mixture.rda"
> load(file(website))
> x <- ESL.mixture$x
> y <- as.factor(ESL.mixture$y)
> plot(x,pch=20, col=ifelse(y==0, "orange", "blue"), asp = 1)
```

The kNN classifier is implemented in the `knn` function. It takes in as arguments the training data matrix, the test data matrix, the training labels, and the neighbour count parameter k. As this is a two-dimensional dataset, we can visualise the decision boundary by computing the classification result on a grid of points.

```
> px1 <- seq(from = range(x[,1])[1], to = range(x[,1])[2], by = 0.1)
> px2 <- seq(from = range(x[,2])[1], to = range(x[,2])[2], by = 0.1)
> grid <- expand.grid(px1, px2)
> points(grid, pch=".")
> head(x)
```

Let us first fit a 5NN classifier and examine the output.

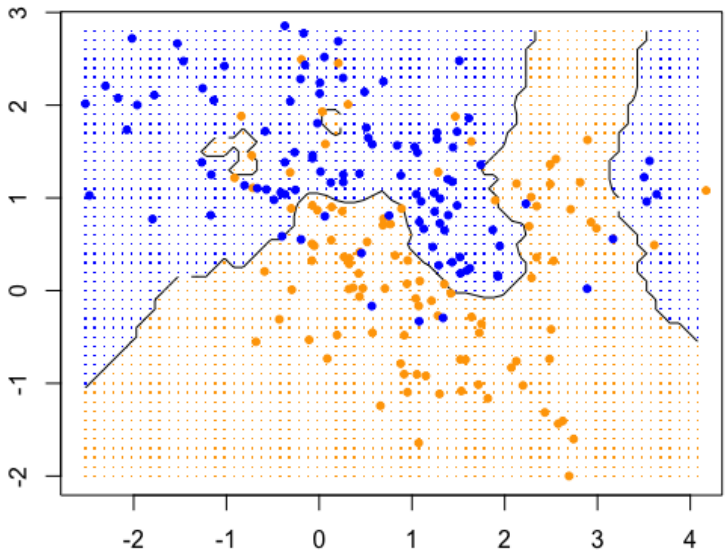
```
> pred <- knn(x, grid, y, k=5, prob=TRUE)
> str(pred)
> attr(pred, "prob")
> p <- ifelse(pred=="1",  attr(pred, "prob"), 1- attr(pred, "prob"))
```

The output `pred` contains a vector of class labels for the test data `grid`, together with three 'attributes' which we can access with the `attr` function. Note that `knn` breaks ties in the majority vote uniformly at random between the most voted classes. Do you understand what is measured by the `prob` attribute? What is the vector `p`? *The `prob` attribute contains the estimated probability that the class label is equal to the predicted label for a given vector of covariates. In other words, it is the proportion of points that have the predicted label in the five nearest neighbours to a given grid point. The vector*

p is the estimated probability that a given grid point belongs to class 1.

We can draw the decision boundary by plotting the 1/2 probability contour from the matrix of probabilities.

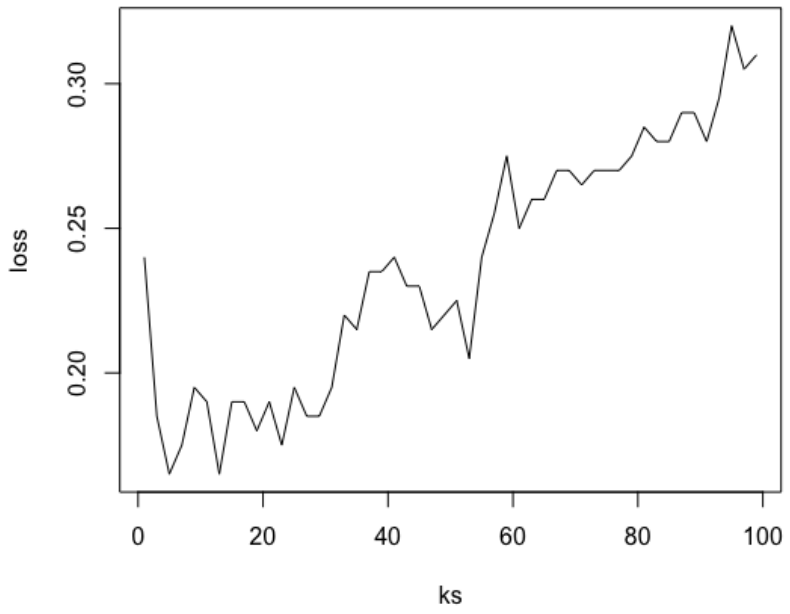
```
> pmatrix <- matrix(p, length(px1), length(px2))
> contour(px1, px2, pmatrix, levels=0.5, labels="", xlab="", ylab="", asp=1)
> points(grid, pch=".", col=ifelse(p<0.5, "orange", "blue"))
> points(x, pch=20, col=ifelse(y==0, "orange", "blue"))
```



Repeat the computations above for $k = 1, 15, 99$. Comment on the resulting decision boundaries. *For $k = 1$, the boundary is very irregular as it must classify every training point correctly (therefore, it will be very sensitive to changes in the data and will probably not generalise well to new test data). As k increases, the decision boundary becomes smoother and less influenced by individual points, but at the same time it is less able to pick out local structure in the signal.*

One way to choose an optimal k is through cross-validation. The `knn.cv` function implements leave-one-out cross-validation although, unlike previous functions performing cross-validation, it does not return the cross-validation error but the predictions for each of the folds. Then, we are free to choose the loss function, and here we choose the $0-1$ loss function by which the classifier has error 1 if it misclassifies the test point at hand and zero otherwise.

```
> ks <- seq(1,99, by=2)
> loss <- rep(0, length(ks))
> for (k in ks){
+     j <- (k+1)/2
+     loss[j] <- sum(knn.cv(x,y,k) != y)/length(y)
+ }
> plot(ks, loss, type="l")
```



The cross-validation error is minimised at $k = 5$ and $k = 13$.

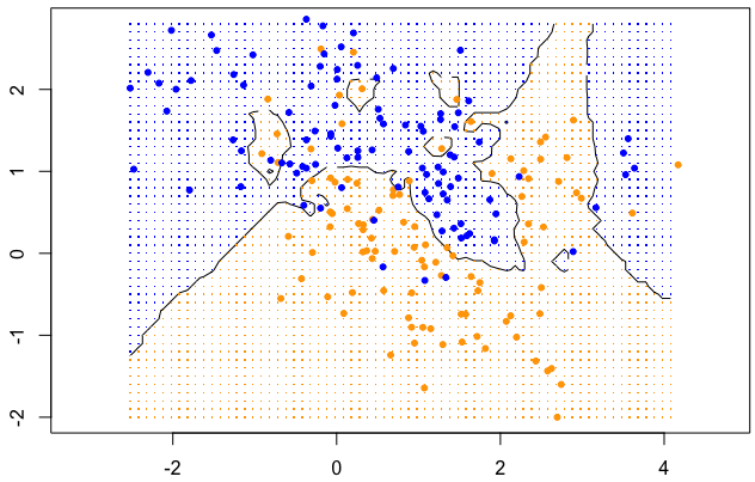
One way to visualise the variability of a classifier is to see how the decision boundary changes on randomly subsampled data points. Here, we draw 9 subsamples from the original dataset and draw the 1NN decision boundaries.

```
> set.seed(2018)
> B <- 9
> m <- 100
> pred <- matrix(0, dim(grid)[1], B)
> old_par <- par()
> par(mfrow=c(3,3), mar=rep(0.5,4))
> for (iter in 1:B){
+     ind <- sample(length(y), m, replace=FALSE)
+     pred[,iter] <- as.numeric(knn(x[ind,], grid, y[ind], k=1)) - 1
+     p <- matrix(pred[,iter], length(px1), length(px2))
+     contour(px1, px2, p, levels=0.5, labels="", xlab="", ylab="", asp=1)
+     points(grid, pch=".", col=ifelse(p<0.5, "orange", "blue"))
+     points(x[ind,], pch=20, col=ifelse(y[ind]==0, "orange", "blue"))
+ }
> par(old_par)
```

What is the subsampling method used above? *Uniform subsampling of size 100 without replacement.*

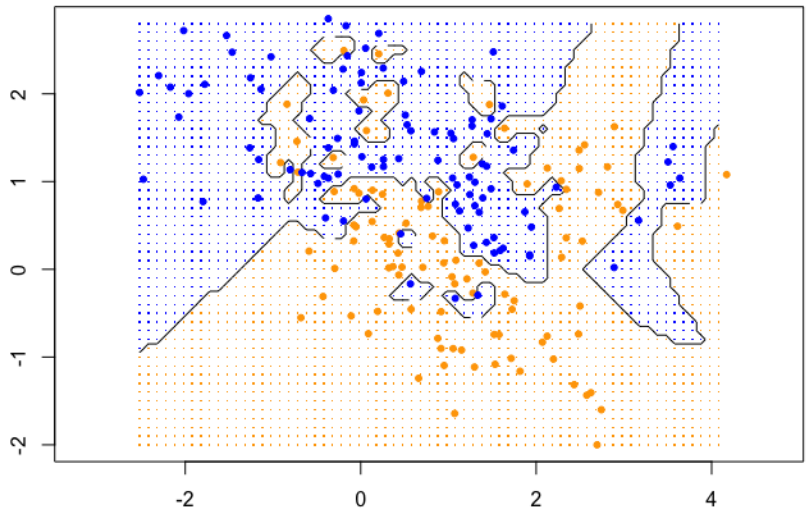
Repeat the computations above (including the `set.seed` instruction) using replacement in the subsampling. We can aggregate the predictions from these 9 subsamples to obtain a bagged nearest neighbour classifier.

```
> p <- matrix(rowMeans(pred), length(px1), length(px2))
> contour(px1, px2, p, levels=0.5, labels="", xlab="", ylab="", asp=1)
> points(grid, pch=".", col=ifelse(p<0.5, "orange", "blue"))
> points(x, pch=20, col=ifelse(y==0, "orange", "blue"))
```



Let us compare this to the 1NN classifier:

```
> pred <- knn(x, grid, y, k=1, prob=TRUE)
> p <- ifelse(pred=="1", attr(pred, "prob"), 1- attr(pred, "prob"))
> pmatrix <- matrix(p, length(px1), length(px2))
> contour(px1, px2, pmatrix, levels=0.5, labels="", xlab="", ylab="", asp=1)
> points(grid, pch=".", col=ifelse(p<0.5, "orange", "blue"))
> points(x, pch=20, col=ifelse(y==0, "orange", "blue"))
```



It seems that the bagged nearest neighbour classifier has not smoothed out the 1NN classifier as much as we hoped. In particular, we can still see the pockets mostly in the upper half of the graph making significant impact on the decision boundary. Why do you think this is the case and how would you adjust the above procedures to bypass it? *We are subsampling 50% of the original training data. Given any test point x, let $X_{(1)}$ be its nearest neighbour in the full training data. The probability that $X_{(1)}$ is included in a random subsample is $1-(1-1/200)^{100} \approx 0.394$. Since we only have 9 bootstrap samples, the probability that $X_{(1)}$ appears in at least 5 out of 9 bootstrap samples is not negligible: it is around 25%, since* `pbinom(4, 9, 0.394)` *is about 0.75. In other words, $X_{(1)}$ still has 25% chance to be classified as its nearest neighbour. If both $X_{(1)}$ and $X_{(2)}$ are of the same class, this figure increases to about* `1 - pbinom(4, 9, 1-exp(-1))`*, which is about 80%. To remedy this problem, we need to decrease the subsampling size so as to reduce the influence of immediate neighbours, and increase the number of bootstrap samples so as to decrease the probability of more than half of the samples containing immediate neighbours. Try the above procedure with $B = 100$ and $m = 25$.*

Optimally weighted kNN classifier. The `ownn` function implements the optimally weighted nearest neighbour classifier and the infinite bootstrap aggregation nearest neighbour classifier with optimal resampling fraction ($\sim n^{-4/(d+4)}$ as $n \to \infty$ under mild assumptions). We compare their performance by setting half of our data for training and the other half for testing.

```
> test = sample(200,100)
> ownn(x[-test,], x[test,], y[-test], y[test])
```

We see that as predicted, both the optimally weighted nearest neighbour classifier and the optimal infinite bootstrap aggregation nearest neighbour classifier performs better than vanilla kNN. However, unexpectedly the bagged classifier seems to outperform the optimally weighted classifier in this example.

MNIST data

We now apply the nearest neighbour classifier to the MNIST data. One major disadvantage of kNN is that the classifier has to compute distance from test data to the entire training data to search for nearest points. As such, kNN can be very computationally intensive. So we will not do cross-validation for the choice of k but rather (arbitrarily) set $k = 9$ in this practical.

```
> filePath <- "https://raw.githubusercontent.com/AJCoca/SLP19/master/"
> fileName <- "mnist.csv"
> mnist <- read.csv(paste0(filePath, fileName), header = TRUE)
> mnist$digit <- as.factor(mnist$digit)
> visualise = function(vec, ...){ # function for graphically displaying a digit
+     image(matrix(as.numeric(vec),nrow=28)[,28:1], col=gray((255:0)/255), ...)
+ }
>
> train <- mnist[1:4000, ]
> test <- mnist[4001:6000, ]
>
> pred = knn(train[,-1], test[,-1], train[,1], k=9)
> actual <- test[,1]
> sum(actual!=pred)/2000
> table(actual, pred)
```

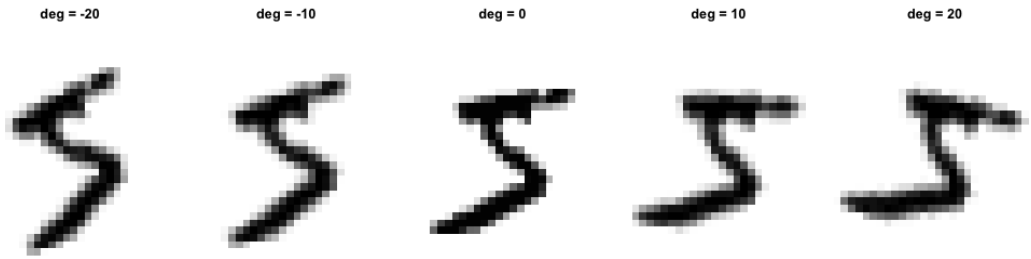
We see that with the simple vanilla kNN classifier, we are already able to achieve a quite impressive 8.15% classification error. One way to improve classification is by augmenting the original dataset with slightly rotated digits (note that this MNIST dataset has been centred; otherwise, we could have also augment it with translated copies). The following non-examinable function achieves the effect of rotating all grayscale images by a certain degree.

```
rotate = function(original, degree){
# rotate 28x28 greyscale images by the given degrees clockwise
rotated = original
img = as.matrix(original[,-1])
ind = matrix(FALSE,40,40); ind[7:34,7:34] = TRUE; ind = as.vector(ind)
tmp = matrix(0, dim(img)[1], 40*40); tmp[, ind] = img
i = rep(7:34, 28); j = rep(7:34, each=28)
a = degree/180*pi
i_rotated = (i-20.5)*cos(a) + (j-20.5)*sin(a) + 20.5
j_rotated = -(i-20.5)*sin(a) + (j-20.5)*cos(a) + 20.5
i0 = floor(i_rotated); i1 = ceiling(i_rotated); lambda_i = i_rotated-i0
j0 = floor(j_rotated); j1 = ceiling(j_rotated); lambda_j = j_rotated-j0
rotated[,-1] = tmp[, i0+(j0-1)*40] * (1-lambda_i) * (1-lambda_j) +
tmp[, i0+(j1-1)*40] * (1-lambda_i) * lambda_j +
tmp[, i1+(j0-1)*40] * lambda_i * (1-lambda_j) +
tmp[, i1+(j1-1)*40] * lambda_i * lambda_j
return(rotated)
}
```

We use the above code to generate rotated images with the following non-examinable code.

```
train10c <- rotate(train, 10)
train20c <- rotate(train, 20)
train10a <- rotate(train, -10)
train20a <- rotate(train, -20)

par(mfrow=c(1,5), mar=c(0.5,0.5,4,0.5))
visualise(train20a[1,-1], main = "deg = -20", asp=1)
visualise(train10a[1,-1], main = "deg = -10", asp=1)
visualise(train[1,-1], main = "deg = 0", asp=1)
visualise(train10c[1,-1], main = "deg = 10", asp=1)
visualise(train20c[1,-1], main = "deg = 20", asp=1)
```



We train the nearest neighbour classifier on the augmented dataset. The resulting prediction error is reduced to 5.55% as the following shows.

```
> train_aug = rbind(train, train10c, train20c, train10a, train20a)
> pred = knn(train_aug[,-1], test[,-1], train_aug[,1], k=9) # this can take long
> actual <- test[,1]
> sum(actual!=pred)/2000
> table(actual, pred)
```

Here is a gallery of misclassified handwritings.

```
> err_ind <- (1:2000)[pred != actual]
> par(mfrow = c(3,3))
> for (i in 1:9){
+     visualise(test[err_ind[i],-1],
+               main=paste0("true=", actual[err_ind[i]], ", pred=", pred[err_ind[i]]))
+ }
> par(old_par)
```