Lab 4: High-dimensional datasets

Biomedical Data Science

Marco Colombo, University of Edinburgh

Correlation plots

We have already seen how to compute a correlation between two numerical vectors. The same function, `cor()`, can be used to compute the correlation of columns of a matrix (or the numerical columns of a dataframe).

```
> diab01 <- read.delim("data/diab01.txt")
> corr <- cor(diab01[, sapply(diab01, is.numeric)], use="pairwise.complete")
```

A good way of visualizing the strength of correlation between the variables is by creating a correlation plot. Package `corrplot` provides a good implementation of such plots.

```
> library(corrplot)
```

The default plot can be improved a bit, for example by not plotting the diagonal elements (they will always be 1) and by using a different font colour for the variable names.

```
> corrplot(corr, diag=FALSE, tl.col="black")
```

Correlation and covariance matrix are better measures of similarity if the data is normal. So often, we try to transform the data so that variables are distributed normally: in most cases, normality can be approximated by taking logs if variables are right-skewed (log-normality) or by using a power transform if left-skewed.

The `diab01` dataset is very small, and besides the strong correlation between LDL and total cholesterol (0.833) there is no strong correlation structure. For a dataset with many more variables, it helps to reorder them so to reveal any block structure that may be present. Function `corrplot()` provides several ordering methods (see `?corrplot` for details): by experience, hierarchical clustering (`order="hclust"`) works well in most cases.

```
> corrplot(corr, order="hclust", diag=FALSE, tl.col="black",
+          title="Correlation matrix (ordered by hierarchical clustering)",
+          mar=c(0,1,2,0))
```

Principal component analysis

We may attempt to run principal component analysis on the `diab01` dataset, even though it's not too large to require dimensionality reduction. Note that PCA makes sense only over numerical column, and that missing values are not allowed. Given the small amount of missingness in `diab01` we will do a quick imputation to the median before proceeding (see function `impute.to.median()` from Lab 2).

```
> diab01.imputed <- diab01   # make a copy to keep the unimputed data around
> for (col in colnames(diab01.imputed))
+   diab01.imputed[, col] <- impute.to.median(diab01.imputed[, col])
```

PCA can be run by using function `prcomp()`. To discover how variables cluster, we need to operate on the transpose of the matrix (obtained by using function `t()`): make sure that the outcome variable is not included in the PCA analysis. Setting option `scale=TRUE` is always advised, as it takes into account the fact that variables may be measured in different units.

```
> idx.pca <- sapply(diab01.imputed, is.numeric) &      # remove factors
+                   colnames(diab01.imputed) != "Y"    # remove outcome
> pca.vars <- prcomp(t(diab01.imputed[, idx.pca]), scale=TRUE)
```

The amount of variability explained by the components can be computed bearing in mind that the square root of the eigenvalues is stored in vector `sdev` of the PCA object. This is always decreasing, and in this instance the first two components account for most of the variability in the data. The variance explained by the principal components can be visualized through a scree plot.

```
> perc.expl <- pca.vars$sdev^2 / sum(pca.vars$sdev^2)
> sum(perc.expl[1:2])
> screeplot(pca.vars, main="Scree plot")
```

The projection of the data on the principal components is given in vector `x` of the PCA object: plotting this can sometimes reveal some interesting patterns in the data.

```
> plot(pca.vars$x[, 1:2], main="Projection of variables on the first 2 PCs")
```

It is also interesting to see what happens if we used PCA to describe the similarity between patients: in this case, we operate on the matrix itself, rather than on the transpose.

```
> pca.pats <- prcomp(diab01.imputed[, idx.pca], scale=TRUE)
> plot(pca.pats$x[, 1:2], main="Projection of patients on the first 2 PCs")
```

The plot doesn't suggest any particular difference among patients.

Using a colour to denote the outcome (as `Y` is continuous, we dichotomize it) doesn't reveal any very clear separation, although possibly for negative values of the two PCs there are more severe cases (high `Y`), while for positive values of both PC there are less severe cases (low `Y`).

```
> y.bin <- with(diab01.imputed, as.integer(Y > median(Y)))
> plot(pca.pats$x[, 1:2], main="Projection of patients on the first 2 PCs",
+      col=1+y.bin, pch=19)
> legend("bottomright", legend=c("Low Y", "High Y"), col=1:2, lwd=4)
```

Subset selection

Stepwise selection can be executed through function `stepAIC()` provided by the `MASS` package (a simplified version, called `step()` doesn't require an additional package).

```
> library(MASS)
```

The function requires an initial model, and adds or removes one predictor at a time (according to what is specified in the `direction` parameter) until no improvement in AIC can be produced.

A special symbol that can be used in formulas is "`.`", which represents all columns in the dataframe. In order to use it, we need to remove the patient identifiers from the columns, otherwise they would be turned into dummy variables and create collinearities.

```
> rownames(diab01.imputed) <- diab01.imputed$PAT
> diab01.imputed$PAT <- NULL
> full.model <- lm(Y ~ ., data=diab01.imputed)       # use all available variables
> sel.back <- stepAIC(full.model, direction="back")  # backward elimination
Start:  AIC=817
Y ~ AGE + SEX + BMI + BP + TC + LDL + HDL + GLU

       Df Sum of Sq    RSS AIC
- GLU   1        89 294900 815
- AGE   1       500 295311 815
<none>              294811 817
- BP    1     13549 308360 819
- BMI   1     16674 311485 820
- SEX   1     19422 314233 821
- TC    1     23329 318140 823
- LDL   1     28736 323547 824
- HDL   1     53638 348449 832

Step:  AIC=815
Y ~ AGE + SEX + BMI + BP + TC + LDL + HDL

       Df Sum of Sq    RSS AIC
- AGE   1       481 295381 813
<none>              294900 815
- BP    1     14728 309628 818
- BMI   1     17857 312757 819
- SEX   1     19518 314418 819
- TC    1     24696 319596 821
- LDL   1     30382 325282 823
- HDL   1     54017 348918 830

Step:  AIC=813
Y ~ SEX + BMI + BP + TC + LDL + HDL

       Df Sum of Sq    RSS AIC
<none>              295381 813
- BP    1     15661 311042 816
- BMI   1     19916 315297 818
- SEX   1     20161 315542 818
- TC    1     27862 323243 820
- LDL   1     33235 328616 822
- HDL   1     53652 349033 828
```

At each iteration of the selection process all attempted models are fitted on the data and their AICs are compared: the chosen model is the one that produces the lowest AIC. Among the models compared there is also the current one (indicated by `<none>`): when this rises to the top, the process stops, as no other model has produced a lower AIC.

In this case, backward elimination stopped after 2 iterations: at the first, variable GLU is removed, and at the second, AGE is removed. After that, the removal of any other variables would cause an increase in AIC, so backward elimination stops.

The object produced by `stepAIC()` is identical to one produced by `lm()` or `glm()`, and it corresponds to the final model. So, for example, to see the results of fitting the model we can use `summary()`.

```
> summary(sel.back)
```

Suppose that we wanted both age and sex in the selected model (these are two of the most common confounders, so usually we want to have them in all models): this can be controlled by the `scope` parameter, which allows to use either a fitted model or just a formula to describe the smallest model allowed (that is, which predictors should never be eliminated).

```
> sel.back <- stepAIC(full.model, scope=list(lower=~ AGE + SEX), direction="back")
```

We could now try forward selection on the same dataset. When going forward, the `scope` parameter must always be specified to indicate the `upper` model, that is which variables should be considered in the selection process (when going backward this is implied by the initial model, which by definition includes all variables of potential interest).

```
> null.model <- lm(Y ~ 1, data=diab01.imputed)          # only include the intercept
> sel.forw <- stepAIC(null.model, scope=list(upper=full.model),
+                     direction="forward")
```

Forward selection chose a sparser model than backward elimination, but this is not guaranteed to happen at all times.

The `step()` function also allows to run stepwise selection by setting `direction="both"`.

```
> init.both <- lm(Y ~ AGE + SEX, data=diab01.imputed)
> sel.both <- step(init.both, scope=list(upper=full.model), direction="both")
```

It is worth remembering that this process identified the model that best fits our dataset: although the backward elimination model has lower AIC than the other models, we do not yet know which model generalises better.

Regularisation approaches

Ridge regression, lasso and elastic net are implemented in the `glmnet` package.

```
> library(glmnet)
```

There are two main functions provided by the package: `glmnet()` and `cv.glmnet()`. The first fits a regularised model for a series of values of the penalty parameter λ (by default 100, but it may be truncated for small datasets); the second, run an internal cross-validation to identify which specific setting of λ performs better when predicting the observations in the test set.

Unfortunately, neither function accepts formulas to define models, but expects matrices and vectors as input. You can use the following function to facilitate the tranformation of a dataframe to a matrix as expected by the `glmnet` package.

```
prepare.glmnet <- function(data, formula=~ .) {

  ## create the design matrix to deal correctly with factor variables,
  ## without losing rows containing NAs
  old.opts <- options(na.action='na.pass')
  x <- model.matrix(formula, data)
```

```
  options(old.opts)

  ## remove the intercept column, as glmnet will add one by default
  x <- x[, -match("(Intercept)", colnames(x))]
  return(x)
}
```

By default, the function uses all existing columns in the dataframe to create However, we do not want the outcome variable to be in the matrix of predictors: so we first remove it, then convert the rest of the dataframe to a matrix.

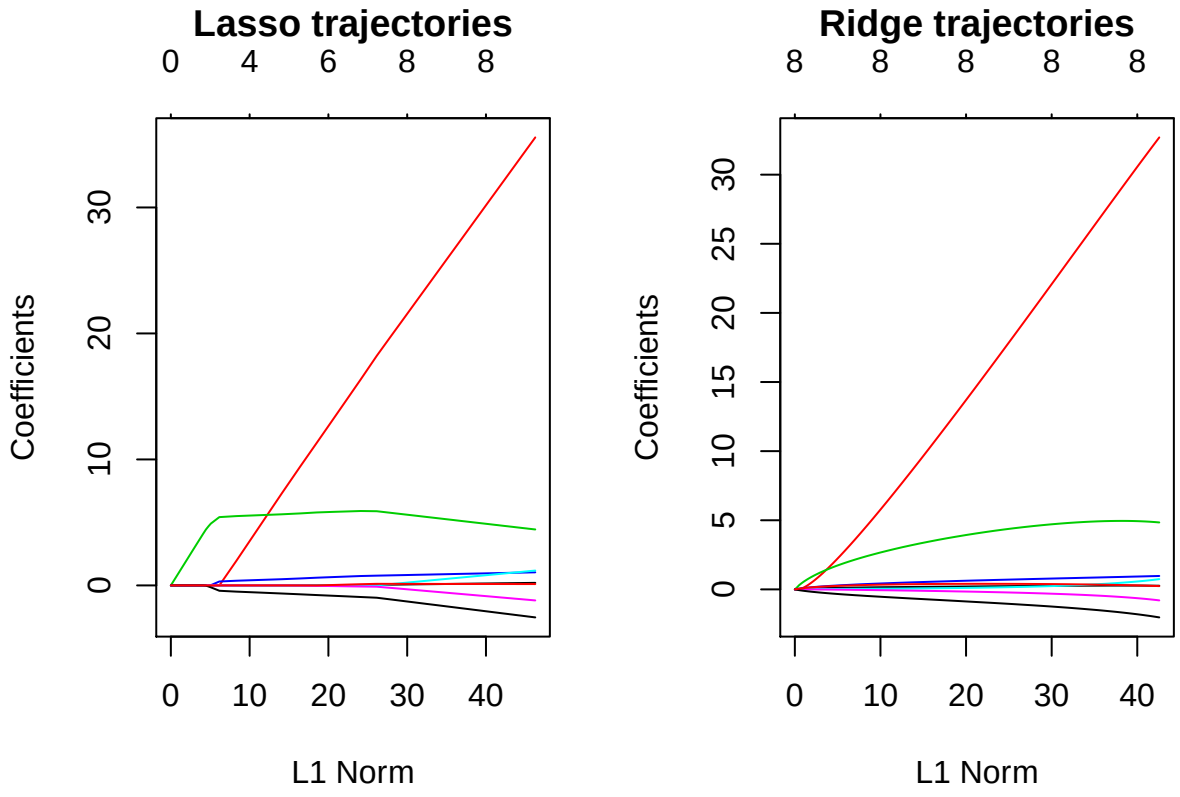
```
> ydiab01 <- diab01.imputed$Y                          # store the outcome separately
> xdiab01 <- prepare.glmnet(diab01.imputed, ~ . - Y) # exclude the outcome
```

Now we are finally ready to fit the first regularised model: by default, function `glmnet()` will fit a linear regression with lasso penalty. To change it to ridge regression, set the `alpha` option to 0.

```
> fit.lasso <- glmnet(xdiab01, ydiab01)           # same as setting alpha=1
> fit.ridge <- glmnet(xdiab01, ydiab01, alpha=0)
```

To see the trajectories of the coefficients for the various choices of λ it's enough to use `plot()` on the fitted objects.

```
> par(mfrow=c(1,2), mar=c(4,4,5,2))
> plot(fit.lasso, main="Lasso trajectories")
> plot(fit.ridge, main="Ridge trajectories")
```



The x-axis indicates the L_1 norm of the regression coefficients: when the penalty parameter λ is at its maximum value all coefficients are zero (null model); by decreasing the strength of the penalty term, the coefficients are allowed to increase. The numbers at the top of the plot count the number of nonzero variables: note how for ridge all predictors become very soon nonzero, while for lasso this happens in a staggered way.

The model coefficients depend on the choice of λ: they can be found in the fields `a0` (for intercepts) and `beta` (for predictors) of the fitted objects, while the value of the corresponding penalty factor is stored in the `lambda` field. Assuming that we were interested in the 10-th value of λ, we could retrieve the corresponding model coefficients by subsetting.

```
> idx <- 10
> lambda10 <- fit.lasso$lambda[idx]
> fit.lasso$a0[idx]                    # intercept
  s9
19.1
> fit.lasso$beta[, idx]                # coefficients
    AGE    SEXM     BMI      BP      TC     LDL     HDL     GLU
 0.0000  0.0000  4.6977  0.0000  0.0000  0.0000 -0.0864  0.0000
```

Note that because of the regularization term, we are not able to produce estimates of the standard error and consequently p-values. There are some recent theoretical results, so maybe in the future they will be computed by functions in the package.

The `predict()` method works in a similar way as for linear and logistic regression. However, unless otherwise specified through the `s` option, the returned values correspond again to all settings of λ. Also, there are a few more types of values that can be obtained through this function (see `?predict.glmnet` for more details).

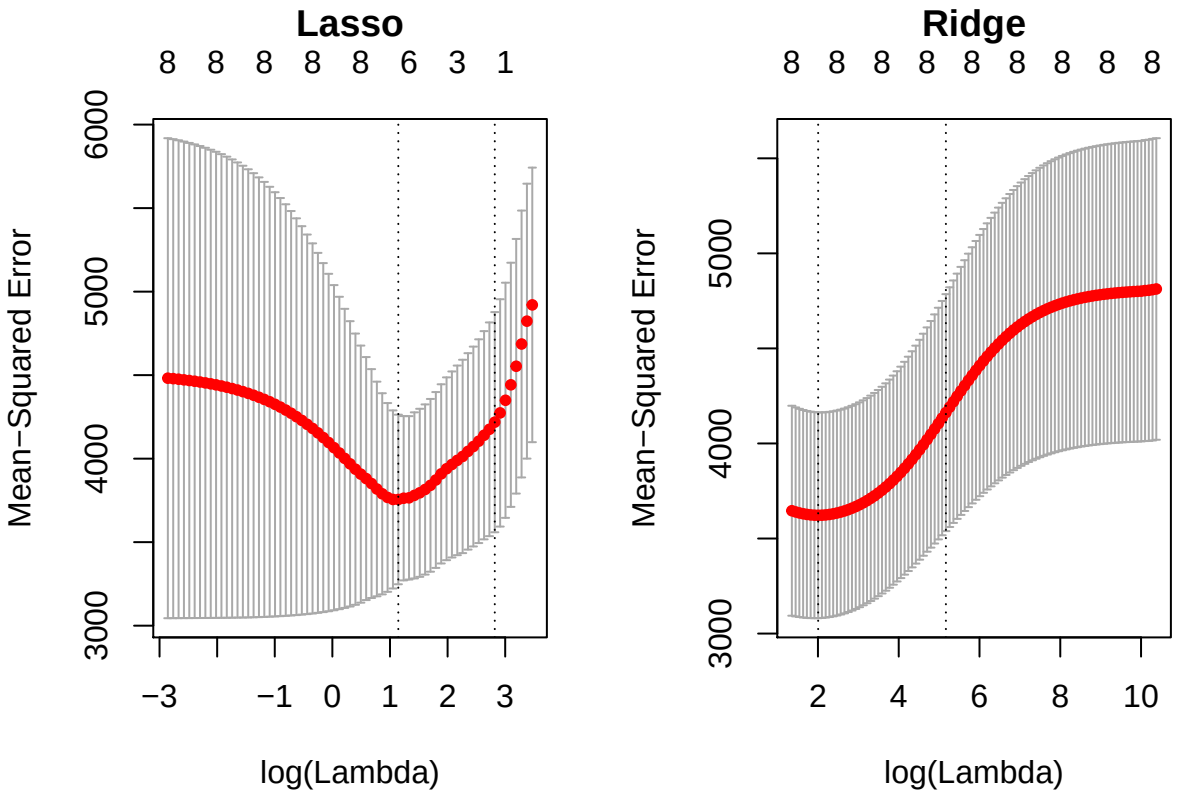
```
> predict(fit.lasso, newx=xdiab01, type="response")
> predict(fit.lasso, newx=xdiab01, type="response", s=lambda10)
> predict(fit.lasso, type="nonzero")  # indices of nonzero elements for each lambda
> predict(fit.lasso, type="coefficients")  # regression coefficients for each lambda
```

In most cases we are interested only in a specific setting of the penalty parameter, ideally one which will be most effective in prediction. To identify it, we can use function `cv.glmnet()` to perform cross-validation: this happens within the function, so we do not need to create a specific set of folds for that (although we may do if we were to learn or tune other parameters).

```
> fit.cv.lasso <- cv.glmnet(xdiab01, ydiab01)
> fit.cv.ridge <- cv.glmnet(xdiab01, ydiab01, alpha=0)
```

Plotting the cross-validation curve allows us to inspect how prediction errors vary according to the amount of shrinkage applied.

```
> par(mfrow=c(1,2), mar=c(4,4,5,2))
> plot(fit.cv.lasso, main="Lasso")
> plot(fit.cv.ridge, main="Ridge")
```



The plot displays the mean cross-validated error in red with bars corresponding to standard errors. The leftmost dotted line in each plot corresponds to the λ that minimizes the error (`lambda.min` in the fitted object); the dotted line to the right corresponds to the largest value of λ such that the error is within one standard error from the minimum (`fit.lasso$lambda.1se` in the fitted object).

The curves obtained depend on the choice of the error measure used in cross-validation. By default, `cv.glmnet()` uses the mean square error for linear regression and deviance for logistic regression. However, these can be changed to mean absolute error (for linear regression) or to AUC or classification error (for logistic regression) by setting the appropriate choice of the `type.measure` option (see `?cv.glmnet`).

Now that we know a good choice of the penalty parameter, we can use that in prediction (although here, since we have hadn't created cross-validation folds, we do not have any new data to predict).

```
> predict(fit.cv.lasso, newx=xdiab01, type="response", s=fit.cv.lasso$lambda.min)
```

Note that inside the object produced by `cv.glmnet()` there is a field called `glmnet.fit` which effectively stores what would have been created by using `glmnet()`: this is where the regression coefficients for all values of λ are stored.

Practical exercises

1. Using the `birthwt.csv` dataset (available on Learn):

   - Build model F1 to predict birth weight (variable "bwt") by using forward selection starting from a model that only includes age of the mother, and report the adjusted R^2 (answer: 0.649).
   - Build model B1 by using backward elimination ensuring that age is never dropped from the model. Write an assertion to test that models F1 and B1 selected the same predictors.
   - What variables are most associated with the outcome? What is the effect of smoking on birth weight?
   - Create a boxplot of birth weight stratified according to variable "low", as well as a scatter plot of birth weight with different colours according to variable "low", to demonstrate the relationship between the two variables.
   - Build a new forward selection model F2 for birth weight ensuring that variable "low" cannot enter the model and report the adjusted R^2 (answer: 0.212).
   - Use an appropriate measure (other than R^2) to compare models F1 and F2 and justify your choice.
   - What is the most strongly associated variable? What is the effect of smoking and can you explain the difference in effect size compared to model F1?
   - Fit a lasso-regularized model for birth weight, ensuring that variable "low" cannot enter the model. Produce a plot of the coefficient trajectories and report the smallest value of the penalty parameter λ for which only 3 predictors have a nonzero coefficient (answer: 118.1643).

2. Using the `clev.csv` dataset (available on Learn):

   - Set the random seed to 1 and create 10 cross-validation folds.
   - Model the occurrence of heart disease in each of the training folds using only age, sex, blood pressure and number of vessels as predictors (hint: see Lab 3).
   - Predict the outcomes for observations in the test folds (hint: see Lab 3). Report the mean cross-validated AUC for the test data (answer: 0.796).
   - In each training fold fit a ridge regression model using the same set of predictors and make a prediction of the outcomes on the test sets when using the optimal $\lambda_{\min}$ found within each fold. Report the mean cross-validated AUC for the test data (answer: 0.797).
   - Looking at the models fitted on the first cross-validation fold, compare the coefficients of the predictors from ridge regression to those from the unpenalised model. Which predictor was penalised least, and which was the most penalised?
   - Build a lasso model using all available predictors apart from "chest.pain" (and the outcome!) following the same cross-validation approach as before. Report the mean cross-validated AUC for the test data (answer: 0.885).
   - Find which predictors are retained at the optimal $\lambda_{\min}$ in the first cross-validation fold and their number (answer: 11).