Exercise 6

Data preparation steps

In predictive modeling, we use an algorithm or *model* to find a relation between some known data, often called *features* or *predictors* and some unknown data, the *target variable*. We have already covered several models like linear and logistic regressions, rpart, k-means clustering. So far you have been using pre-processed data sets. It is time to learn how to go around your variables. We will start with the learning the tricks to transform a pile of numbers into a crisp ready-for-modelling data set and proceed to expand the borders, learning to tell a usefull variable predictor from noise and create entirely new variables that will advance our models beyond generic design.

Loading, structural checks

Let's work through a typical data preparation problem by preparing the *loans* dataset for regression. 1. Load the .csv data file. Be careful to specify the correct formatting and do not use the helper_function, we don't need it anymore. 2. Look at the structure of the data and the summary statistics of the numeric variables. Also compare the data to the data dictionary to identify any issues that need treatment. 3. Now look at the values that the nominal variables take. Check how many unique values that the text variables take and, if not too many, create a *table* telling you how often each level occurs.

```
# Load the data
# Be careful to correctly specify the format of the data
loans <- read.csv("loan_data.csv", sep = ";", stringsAsFactors = FALSE)

# Check the data structure and the individual variables
str(loans)
```

```
## 'data.frame':    1225 obs. of  15 variables:
##  $ YOB    : num  19 41 66 51 65 42 59 43 52 65 ...
##  $ nKIDS  : num  4 2 0 2 0 2 0 1 0 0 ...
##  $ nDEP   : num  0 0 0 0 0 0 0 0 0 0 ...
##  $ PHON   : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ dINC_SP: num  0 0 0 0 0 10500 6500 13500 0 0 ...
##  $ EMPS_A : chr  "R" "P" "N" "P" ...
##  $ dINC_A : num  0 36000 30000 464 15000 48000 30000 9000 22500 19500 ...
##  $ RES    : chr  "O" "O" "N" "O" ...
##  $ dHVAL  : num  14464 0 0 24928 0 ...
##  $ dMBO   : num  4 0 0 8464 0 ...
##  $ dOUTM  : num  0 280 0 584 0 1120 520 0 0 540 ...
##  $ dOUTL  : num  0 664 0 320 0 0 0 200 200 0 ...
##  $ dOUTHP : num  0 0 0 0 0 0 96 0 0 0 ...
##  $ dOUTCC : num  0 80 0 60 0 0 0 0 80 0 ...
##  $ BAD    : num  0 0 0 0 0 0 0 0 0 0 ...
```

```
# Check the summary statistics
summary(loans)
```

```
##       YOB            nKIDS             nDEP              PHON       
##  Min.   : 3.00   Min.   :0.0000   Min.   :0.00000   Min.   :0.0000  
##  1st Qu.:42.00   1st Qu.:0.0000   1st Qu.:0.00000   1st Qu.:1.0000  
##  Median :55.00   Median :0.0000   Median :0.00000   Median :1.0000  
##  Mean   :51.04   Mean   :0.6237   Mean   :0.03837   Mean   :0.9037  
##  3rd Qu.:63.00   3rd Qu.:1.0000   3rd Qu.:0.00000   3rd Qu.:1.0000  
##  Max.   :99.00   Max.   :5.0000   Max.   :2.00000   Max.   :1.0000
```

```
##     dINC_SP          EMPS_A              dINC_A          RES
##  Min.   :    0   Length:1225        Min.   :    0   Length:1225
##  1st Qu.:    0   Class :character   1st Qu.: 9000   Class :character
##  Median :    0   Mode  :character   Median :19500   Mode  :character
##  Mean   : 1990                      Mean   :21244
##  3rd Qu.: 1040                      3rd Qu.:30600
##  Max.   :50000                      Max.   :64800
##      dHVAL            dMBO           dOUTM          dOUTL
##  Min.   :    0   Min.   :    0   Min.   :   0   Min.   :    0.0
##  1st Qu.:    0   1st Qu.:    0   1st Qu.:   0   1st Qu.:    0.0
##  Median :    0   Median :    0   Median : 256   Median :    0.0
##  Mean   :15694   Mean   :11226   Mean   : 342   Mean   :  121.9
##  3rd Qu.:28928   3rd Qu.:20000   3rd Qu.: 528   3rd Qu.:    0.0
##  Max.   :64928   Max.   :64000   Max.   :3800   Max.   :28000.0
##      dOUTHP            dOUTCC            BAD
##  Min.   :   0.00   Min.   :   0.0   Min.   :0.0000
##  1st Qu.:   0.00   1st Qu.:   0.0   1st Qu.:0.0000
##  Median :   0.00   Median :   0.0   Median :0.0000
##  Mean   :  28.72   Mean   :  39.6   Mean   :0.2637
##  3rd Qu.:   0.00   3rd Qu.:   0.0   3rd Qu.:1.0000
##  Max.   :1600.00   Max.   :2800.0   Max.   :1.0000
```

```
# Unique value of character variables
str(unique(loans$EMPS_A))
```

```
##  chr [1:11] "R" "P" "N" "E" "B" "V" "T" "U" "W" "M" "Z"
```

```
str(unique(loans$RES))
```

```
##  chr [1:5] "O" "N" "P" "F" "U"
```

```
# Level counts
table(loans$EMPS_A)
```

```
##
##   B   E   M   N   P   R   T   U   V   W   Z
##  30 124  23   6 531 104 123   8 231  37   8
```

```
table(loans$RES)
```

```
##
##   F   N   O   P   U
## 129  66 624 252 154
```

Converting and adjusting variables

There are levels with very few values in variable *EMPS_A*. Check their meaning in the dictionary and, if plausible, combine them into a new category. Try to develop brief code that doesn't require you to copy and paste the same line several times.

```
# Recombine rare variable levels
# Don't do loans$EMPS_A[loans$EMPS_A == "U"] <- "O" three times! Typos happen and changes will be painf
# A nicer solution would be, for example,
loans$EMPS_A <- ifelse(loans$EMPS_A %in% c("U", "N", "Z"), "O", loans$EMPS_A)
#Check the number of levels and make sure there is no more than 20 as some models will not be able to "
```

Dealing with nominal/categorical variables

Because we need to multiply and sum up values, the input that we need for the model to work is a matrix (think excel table) of numbers without any text data or missing information. Consequently, the first step in building a model starts with preparing the data. In practice, preparing the data means finding and fixing the mistakes that were made when designing and filling the database and (creatively) translating and breaking down information into pieces that the model understands and can work with easily. We will drop the loan dataset for a moment and take a look at a little example.

While people like to work with words and categories, these need to be transformed to numbers in some way in order to fit into most algorithms. If people enter into a database that their hometown is Berlin, Hamburg or Leipzig, we need a numerical transcription of this information. The standard way to do this is called *dummy encoding*. Instead of one variable **hometown** that contains the city names, the same information can be saved in three variables **Berlin**, **Hamburg**, **Leipzig** that take the value 1, if the person is from the respective city and 0 if the person is from another city. You'll notice the disadvantage that the full set of possible values needs to be known, which is usually solved by a variable **Other**.

Person	Hometown	Berlin	Hamburg	Leipzig	Other
A	Berlin	1	0	0	0
B	Leipzig	0	0	1	0

R has a special class *factor* to make working with categorical variables more convenient by turning them into the above format only when necessary. Function **data.frame()** automatically turns character vectors into factor vectors.

How does this work? Saving a string in memory takes up more space than saving a number. To save memory, R assigns a number to every unique string in the vector. It then saves the original vector as a vector of these numbers and remembers which number corresponds to which word, like using student IDs instead of full names and birth dates.
Example: Imagine a column of town names, for example as part of an address. As a character vector it looks like this:
(Berlin, Berlin, Hamburg, Leipzig, Stuttgart, Hamburg, Hamburg, Stuttgart)
Saving the names as strings takes a lot of memory and is inconvenient for other functions. So we transform it to a factor vector that looks like this: (1, 1, 2, 3, 4, 2, 2, 4)
and keep track of which number matches which city:
1 = Berlin, 2 = Hamburg, 3 = Leipzig, 4 = Stuttgart

Note: By default the levels are re-ordered (alphabetically) in increasing order.

When a factor variable is used in a model formula, then R will automatically transform it into the above 1/0 matrix behind the scenes. So be careful when working with many factor levels!

```
# Create a character vector containing the home towns of 8 persons
cities <- c("Hamburg", "Berlin", "Berlin", "Leipzig", "Stuttgart", "Hamburg", "Hamburg", "Stuttgart")
cities
```

```
## [1] "Hamburg"   "Berlin"    "Berlin"    "Leipzig"   "Stuttgart" "Hamburg"
## [7] "Hamburg"   "Stuttgart"
```

```
str(cities)
```

```
##  chr [1:8] "Hamburg" "Berlin" "Berlin" "Leipzig" "Stuttgart" "Hamburg" ...
```

```
# Transform character to factor vector
cities_factor <- factor(cities)
cities_factor
```

```
## [1] Hamburg   Berlin    Berlin    Leipzig   Stuttgart Hamburg   Hamburg
```

```
## [8] Stuttgart
## Levels: Berlin Hamburg Leipzig Stuttgart
```

```
str(cities_factor) # Note that the values are now 2 1 1 3 ...
```

```
##  Factor w/ 4 levels "Berlin","Hamburg",..: 2 1 1 3 4 2 2 4
```

When you use factors in functions, R will usually give you output that is much more convenient for analysis. But keep in mind that it is good practice to assign convenient names to the factor levels. To be on the safe side, they shouldn't be numbers or contain special characters (including space) when avoidable.

```
# Create a numeric vector with the street number of 8 persons
# The vector class is numeric, even though the information is categorical
street_number <- c(18, 25, 23, 11, 14, 10, 12, 14, 15, 16)
street_factor <- factor(street_number)
str(street_factor) # You can see how this can get confusing
```

```
##  Factor w/ 9 levels "10","11","12",..: 7 9 8 2 4 1 3 4 5 6
```

```
# Warning: Transforming 'number factors' back to numbers is not straight-forward
as.numeric(street_factor)
```

```
##  [1] 7 9 8 2 4 1 3 4 5 6
```

```
as.numeric(levels(street_factor))[street_factor]
```

```
##  [1] 18 25 23 11 14 10 12 14 15 16
```

```
# Create a binary vector (0/1) specifying if the address is an apartment
apartment <- c(1, 0, 1, 0, 1, 1, 1, 0)
# We can automatically assign new 'names' for the factor levels
# WARNING: The original levels are re-sorted in increasing order
#          Either sort the labels vector by the new order
factor(apartment, labels = c("house", "apartment"))
```

```
## [1] apartment house     apartment house     apartment apartment apartment
## [8] house
## Levels: house apartment
```

```
#          Or specify the levels explicitly
factor(apartment, levels = c(1, 0), labels = c("apartment", "house"))
```

```
## [1] apartment house     apartment house     apartment apartment apartment
## [8] house
## Levels: apartment house
```

```
# Do not do this:
apartment_naive <- factor(apartment)
as.numeric(apartment_naive)
```

```
## [1] 2 1 2 1 2 2 2 1
```

```
# Can you explain why the 0/1 vector turned to 1/2?
```

Transformation of factor variables

How R deals with factor variables depends on the model and its implentation. Some algorithms can deal with factor variables directly (e.g. decision trees), other algorithms will transform them to a numeric matrix using one-hot-encoding. One-hot-encoding creates a matrix with a binary indicator variable for each level of the original factor vector. Be careful, these matrices become huge for vector with many levels!

```r
# One-hot-encoding in R
model.matrix(~cities)
```

```
##   (Intercept) citiesHamburg citiesLeipzig citiesStuttgart
## 1           1             1             0               0
## 2           1             0             0               0
## 3           1             0             0               0
## 4           1             0             1               0
## 5           1             0             0               1
## 6           1             1             0               0
## 7           1             1             0               0
## 8           1             0             0               1
## attr(,"assign")
## [1] 0 1 1 1
## attr(,"contrasts")
## attr(,"contrasts")$cities
## [1] "contr.treatment"
```

There are two character variables that we cannot put into a model as-is. Why was that again? What was the standard solution mathematically speaking? 1. Transform the two character variables into factor variables. Again, don't copy/paste the same line two times, but find a way to automatize the process. 2. Variable **BAD** specifies if an applicant has defaulted on their loan. It will be convenient later for some functions to transform this variable to a factor variable. Transform **BAD** to factor and relabel it so that "good" indicates no default (0) and "bad" indicates a default (1). 3. Variable PHON seems to be binary. Phone number was either provided or not. Check for missing values and make a decision about converting it to factors.

```r
# There are as usual many ways to do this efficiently
for(chrVar in c("EMPS_A", "RES")){
  loans[, chrVar] <- factor(loans[[chrVar]])
}
# or with even more automation
chrIdx <- which(sapply(loans, is.character))
loans[, chrIdx] <- lapply(
  loans[, chrIdx],
  factor
)

# Convert BAD to factors and relabel its levels
loans$BAD <- factor(loans$BAD, levels=c(0,1), labels=c("good", "bad"))

#Phone number
table(loans$PHON)
```

```
##
##    0    1
##  118 1107
```

```r
loans$PHON <- as.factor(loans$PHON)
```

Weight of Evidence (WoE)

There are more advanced methods for getting rid of too many factor levels, one of them is **Weight of Evidence**, described in the lecture. It is a supervised (i.e. based on the target variable) approach to project a categorical variable with many levels onto one numeric variable. The general idea is to replace each level by a numeric value indicating which class this level more likely to be associated with. Since this is a supervised technique, overfitting is a potential problem. It is advisable to split a woe training set from the overall

training data to calculate the woe values. Because the loans data set is very small, we will refrain from doing so in this exercise.

1. Calculate weight of evidence for variables **EMPS_A** and **RES** on the **woe.data** using function **woe()** from package **klaR**. Remember that the formula for the woe of each factor level is

$$woe_{level} = ln\left(\frac{p(BAD)_{level}}{p(GOOD)_{level}}\right).$$

   Because both the denominator and the argument to the natural log cannot be 0, the math will require at least one observation for each combination of level and target category to work. Use argument **zeroadj = 0.5** to correct with an *additive constant to be added for a level with 0 observations in a class* (see R help).
2. Plot the weight of evidence for the levels of **EMPS_A**. It is always advisable to check created data for plausibility. Do the weights generally fit your expectations?

```
#install.packages("klaR")
library(klaR)
# Calculating the WoE based on a part of the data separate from the training and test data is advisable
# In practice, we use more data for the actual training set and less for the WOE set (but enough so tha
# or maybe even use the full in-sample for both, model training and WOE estimation

# Manual example for a single variable
# Calculate the probability for each target classes within each of the categories of a the variable
# Package {data.table} enhances the base data.frame format for efficient computions including group-wis
classCounts <- table(loans$BAD, loans$EMPS_A)
classProb <- classCounts / rowSums(classCounts)
EMPS_A_woe <- log(classProb[2,] / classProb[1,])
EMPS_A_woe
```

```
##           B           E           M           O           P           R
##  0.01536129  0.13314432 -0.01449168  0.65923742 -0.22580077  0.91144931
##           T           V           W
##  0.18353281 -0.41995679  0.75502848
```

```
# Check the occurrence of the levels in both classes
# Since the data set is small, not all levels may occur for both classes,
# which is a potential problem for the calculation since it leads to zeros
# in the fraction calculation and ln()
tapply(loans$EMPS_A, loans$BAD, summary)
```

```
## $good
##   B   E   M   O   P   R   T   V   W
##  22  88  17  13 413  55  86 187  21
##
## $bad
##   B   E   M   O   P   R   T   V   W
##   8  36   6   9 118  49  37  44  16
```

```
# A way to avoid this is to add a small value to the group counts that are zero

# Use function woe() to calculate the weight of evidence for each factor level
woe.object <- woe(BAD ~ ., data = loans, zeroadj = 0.5)
# Note that sometimes you will see a warning message telling you about empty cells
# It is safe to ignore these messages as we set the parameter
# zeroadj above
```

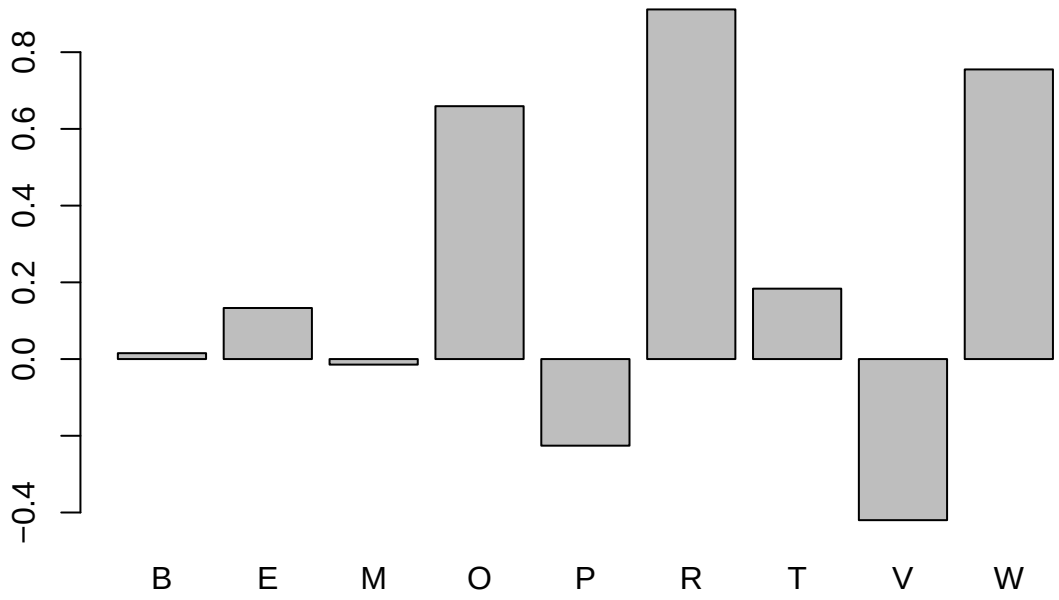
```r
# The weights for each factor level are saved in list element 'woe'. NOte how the function proccessed a
woe.object$woe
```

```
## $PHON
##           0           1
## -0.28252172  0.03228902
##
## $EMPS_A
##           B           E           M           O           P           R
## -0.01536129 -0.13314432  0.01449168 -0.65923742  0.22580077 -0.91144931
##           T           V           W
## -0.18353281  0.41995679 -0.75502848
##
## $RES
##            F            N            O            P            U
##  0.166960272 -0.905601340  0.004397865  0.136188613  0.089041835
```

```r
# This displays the woe per original level of a variable. Using this information, you are able to apply
# the WOE transformation to novel data and the observations in the test set in particular.
# Note that klaR assumes the first level of the target variables to be the target level
# so that the direction of the weights refers to predicting 'good' risks

# The object returned by the woe function also contains a transformed
# version of the data set, where categorical variables have been replaced
# with their woe
#summary(woe.object$xnew)

# The following is more convenient:
# We can apply the woe transformation
# to a data frame with function predict()
# This seems unintuitive, but technically predict() just applies some model
# trained on one data set to another data set and this is what we want to do
# with our woe 'model'
#Let's create a fake test set
test <- loans[sample(1:nrow(loans),80,
    replace=FALSE),]
test.woe <- predict(woe.object, newdata = test, replace = TRUE)
```

```
## No woe model for variable(s): BAD
```

```r
#summary(test.woe)

# Check for plausibility by plotting the weights against their levels
# Remember that the direction is switched
barplot(-1 * woe.object$woe$EMPS_A)
```




```
# Increased default risk seems to apply to the retired, unemployed, house partners and non-respondants
# within the data (and time). This seems intuitive.
```

Treatment of missing values

Variable Year of Birth (**YOB**) contains missing values. As we can see from the data dictionary, these are coded as *99*. Note that it is bad practice to code missing observations as values that can be easily missed. Still, it happens a lot in practice. In R, missing values are denoted as NA (without quotes). While some models can work around missing values, most models will drop the observation completely by default. Let's see if we can do something about this.

1. Replace the value 99 in variable **YOB** with *NA*. You can assign *NA* to a variable like any other value.
2. The missing values could be the result of a random process (e.g. errors in data input) or be more systematic. For example, some people might prefer to not state their age and will leave the field *Year of Birth* empty on their application. To sustain this information, create a dummy variable **YOB_missing** that is 1 if **YOB** *is NA* and 0 otherwise.
3. There seems to be no clear way to infer the missing values in variable **YOB** (year of birth) from other data. Since there are few missing values, we could consider deleting the corresponding applicants (rows) from our data. Alternatively, we will replace them by the mode (= most frequent value) for the variable. HINT: Sort the value count of **table()** to find the most common value quickly. After you are done, drop the YOB_missing value to keep your dataset clean.

```
# Replace the missing observations with NA to make it explicit that they are missing
# R will treat these observations differently from normal values
loans$YOB[loans$YOB == 99] <- NA
summary(loans$YOB)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
```

```
##    3.00   42.00   55.00   50.76   63.00   69.00       7
```

```
# Create a dummy variable YOB_missing with 1 when YOB is na and 0 elsewise
# Do not use '== NA', since NA is not a normal variable value. Instead, use function is.na
loans$YOB_missing <- ifelse(is.na(loans$YOB), 1, 0)
#Check your results by printing the first six values using function head
head(loans$YOB_missing)
```

```
## [1] 0 0 0 0 0 0
```

```
## Find the most frequent value
# Function table gives a vector with the count of all unique values of YOB (second line) and the names
sort(table(loans$YOB))
```

```
##
##  3 17  7  9 10 12 13 14 16  8 15 20 23 25 11 21 22 24 27 28 34 36 31 32 18
##  1  2  3  3  3  3  3  3  3  4  4  4  4  5  6  7  7  7  8  8  9  9 10 10 11
## 19 29 26 35 33 38 39 30 41 45 37 43 40 42 44 47 49 51 53 54 48 50 46 60 56
## 11 12 13 13 15 15 15 16 16 16 17 17 18 20 20 21 23 23 23 24 25 25 26 31 33
## 52 55 57 61 58 59 67 65 69 68 62 63 66 64
## 36 37 37 37 39 40 42 43 45 46 47 47 47 50
```

```
countYOB <- table(loans$YOB)
# Print the names, i.e. unique variable values, and select the value(s) were the count is equal to the
names(countYOB)[countYOB == max(countYOB)]
```

```
## [1] "64"
```

```
# The most frequent year of birth is '64.
# Replace all missing values NA with 64
loans$YOB[is.na(loans$YOB)] <- 64
#It is your call to change the type of YOB variable. If you keep it as numeric, the models will be able
loans$YOB_missing <- NULL
```

Outlier treatment

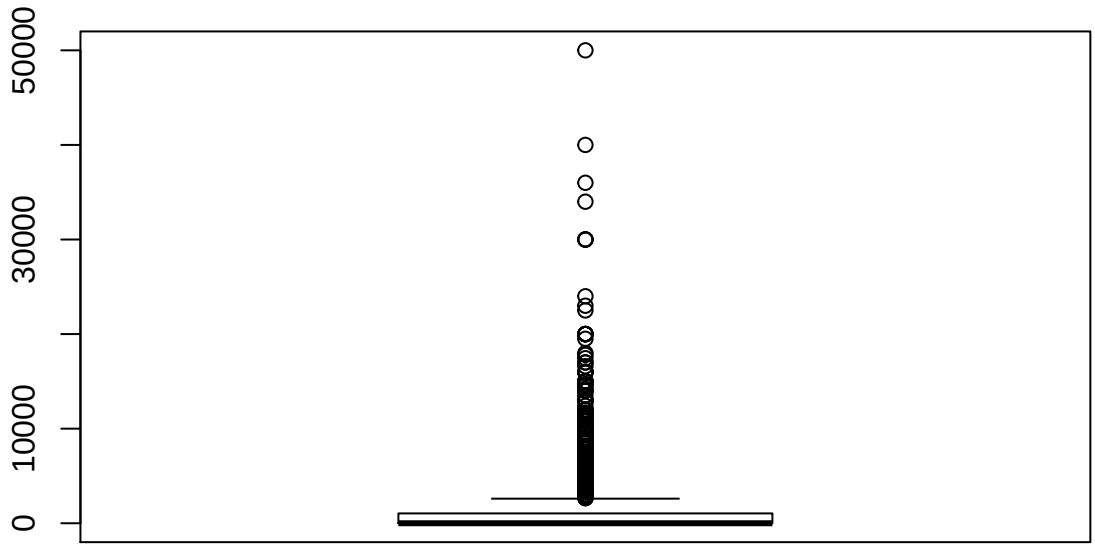
Outliers are variable values that are substantially different (larger or smaller) than other observed values and may have a large impact on the model. They can occur naturally (someone just earns millions or nothing) or be an artifact of the data collection, e.g. errors (999) or wrong input (income of -1000). Depending on our understanding of the data, we can drop outliers, delete erroneous values or truncate them to control their impact on model estimation. Note that many of the more sophisticated prediction models are robust against outliers in the predictors, so that extensive treatment of outliers may give rather little benefit in practice.

For the sake of the exercise, let's assume that we are concerned about outliers in the spouse's income. 1. Check the concern with a boxplot of variable **dINC_SP 2. Create a small function to calculate the z-score of a variable. This is, in fact, the variable standardized by substracting the mean and dividing by the standard deviation. Calculate the z-scores and save them to a variable** zScores**. 3. Replace all values in** dINC_SP** that lie above a z-score of three by the threshold value three standard deviations above the mean.

```
# Boxplot of spouse's income
summary(loans$dINC_SP)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0       0       0    1990    1040   50000
```

```
boxplot(loans$dINC_SP)
```




```
# Calculate the z-score with the function we created earlier for standardization
standardize <- function(x){
    # The actual calculations necessary to standardize the values
    mu <- mean(x)
    std <- sd(x)
    result <- (x - mu)/std
    # Return ends the function and outputs the result
    return(result)

    # A list can be returned to output more than one result
    #return(list("mean" = mu, "sd" = std, "output" = result))
}

# Calculate and save the z-scores
zScores <- standardize(loans$dINC_SP)
summary(zScores)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -0.4144 -0.4144 -0.4144  0.0000 -0.1978  9.9972
```

```
# Replace the assumed outliers with a threshold value
loans$dINC_SP[zScores > 3] <- mean(loans$dINC_SP) + 3*sd(loans$dINC_SP)
```

Simple feature engineering

The structure of the model restricts how the algorithm can work with the data. In the linear regression case, for example, the assumption that the effects are linear does not allow the model to capture non-linear effects.

For example, if a person's income tends to be higher if their spouse earns little and also if their spouse earns much, but not if their spouse earns an average income, then a linear model like logistic regression will not be able to pick up such non-linear pattern.
An efficient way to improve predictive performance is therefore to break down the information contained in the predictors to make them easier for the model to work with. In the regression case, for example, it is very common to use the square of variables as additional predictors or to add the interactions between variables. 1. Add variables that capture the squared income of the spouse as well as the interaction of the home value and outstanding mortgage. A simple interaction term is just the product of two variables. 2. Replace the employement variable **EMPS_A** by a variable that captures the mean income in each employment group using function **ave()**. This is an excellent way to make information explicit to the algorithms when making predictions. Be careful to calculate averages on past data at the time of the observation and the training data only!

```
# New variables to include non-linearity
loans$dINC_SP2 <- loans$dINC_SP ^ 2
loans$house_mortgage <- loans$dHVAL * loans$dMBO

# Average income by group
loans$EMPS_A <- ave(loans$dINC_A, loans$EMPS_A, FUN = mean)

# In practice, using packages from the {tidyverse} or {data.table} is a lot faster!
```