Overview of the Data Mining Process
Part 2: Data Pre-processing

Aram Balagyozyan

Department of Operations and Information Management
Kania School of Management
The
University of Scranton

February 13, 2022

Outline

1. Data Pre-Processing

Handling Categorical Variables

- If the categorical variable is ordered (age group, degree of creditworthiness, etc.), we can code the categories numerically (1, 2, 3,. . .) and treat the variable as if it was a continuous variable.
- Nominal (unordered) categorical variables, however, often can be treated as is. In many cases, they must be decomposed into a series of binary variables, called *dummy variables*.
- For example, column REMODEL in our West Roxbury housing data contains 3 distinct values indicating whether the house was remodeled recently, long time ago, or never. Those indicator values are: "Recent," "Old," and "None" respectively. We could split this column into three dummy variables
  1. REMODEL RECENT - Yes(1)/No(0)
  2. REMODEL OLD - Yes(1)/No(0)
  3. REMODEL NEVER - Yes(1)/No(0)

Handling Categorical Variables (continued)

- Note that if we were to split the REMODEL variable into binary variables, we would need only two (not three) of them, because if the house was remodeled either recently or long time ago, then we would know that REMODEL NEVER should be $=$ 0. Alternatively, if REMODEL RECENT $= 0$ and REMODEL OLD $= 0$, then REMODEL NEVER should be $= 1$.
- The above detail is important because some analytic routines (e.g. linear or logistic regression) would fail if you use not two but three dummy variables.

Handling Categorical Variables (continued)

- In order to create a dummy for REMODEL use the `model.matrix()` command:

```
xtotal<-model.matrix(~0+REMODEL, data = housing.df)
# convert the xtotal matrix into a data frame
xtotal<-as.data.frame(xtotal)
# check the first 2 rows
head(xtotal,2)
```

```
##   REMODELNone REMODELOld REMODELRecent
## 1           1           0             0
## 2           0           0             1
```

```
# drop REMODELNone
xtotal<-xtotal[,-1]
# Join xtotal with housing.df after droping
# column REMODEL from it.
housing.df<-cbind(housing.df[,-14],xtotal)
```

Variable Selection: How Many Variables and How Much Data

- In general, compactness (aka parsimony) is a desirable feature in a data mining model. Thus, more is not necessarily better when it comes to selecting variables for a model.
- The more variables we include in the model and the more complex the model, the greater the number of records we will need to assess relationship among the variables.
- Even when we start with a small number of variables, we often end up with many more after creating new variables (such as converting a categorical variable into a set of dummy variables). Data visualization and dimension reduction methods help reduce the number variables so that redundancies and information overlap are reduced.
- A good rule of thumb is to have 10 records (observations) for every predictor variable. For classification problems, Delmaster and Hancock (2001) suggest to have at least $6 \times m \times p$ records, where m is the number of outcome classes and p is the number of variables.

Variable Selection: How Many Variables and How Much Data (continued)

- Even when we have ample of data, there is a good reason to pay close attention to the variables in the model.
- Someone with domain knowledge should be consulted, as knowledge of what the variables represent is typically critical for building a good model.
- Putting domain knowledge aside, there are many analytic methods that we can use to select features. These frequently cast into
  i. **filter methods**: involve looking at variables individually and asserting their value using some metric, which is then used to rank them and remove the less relevant ones.
  ii. **wrapper methods**: search for the subset of variables that are more adequate in terms of criteria used to evaluate the results of the posterior modeling stages. In this context, the candidate models are brought back to the feature selection step so that we can select the subset of variables that best optimize their performance.

Variable Selection: How Many Variables and How Much Data (continued)

- As a simple examples of the filter method, we often calculate the correlation between the target variable and each feature in a set. Some other examples include for instance, checking for constant features (i.e. variables that have constant value on all observations and, therefore, are useless for prediction), or for ID-like variables (i.e. variables that are different on all observations).
- For details on specific filter methods, refer to the documentation of packages **FSelector** (Romanski and Kotthoff, 2014) and **CORElearn** (Robnik-Sikonja and Savicky, 2015) that implement many of the metrics that can be used for feature selection.

Variable Selection: How Many Variables and How Much Data (continued)

- Wrapper methods typically involve an iterative search procedure where at each step a candidate set of features is used to obtain a model, which is evaluated and the results of this evaluation are used to decide if the features are good enough or if we need to try other set.
- Examples of wrapper methods include tree-based models that carry out some feature selection as part of the modeling stage, or even tools that can be used to estimate the relevance of each feature for their modeling approach (we will see examples of this with random forests later in the course).

Handling Dates

- Dates are becoming more and more common in data sets. With the existence of a wide range of formats for storing a date, converting between these formats or extracting information from the provided values turns out to be a frequent task we need to carry out during data pre-processing.
- Package **lubridate** (Grolemund and Wickham, 2011) is particularly handy in terms of parsing different date and time formats, as well as extracting different components of these dates.
- Lubridate functions have names composed by the letters "y", "m", "d", "h", "m" and "s" arranged in a way to match the format of the string you are trying to parse.

Handling Dates (continued)

- Below are few examples of lubridate functions.

```
library(lubridate)
ymd("20151021")
```

```
## [1] "2015-10-21"
```

```
ymd("2015/11/30")
```

```
## [1] "2015-11-30"
```

```
myd("11.2012.3")
```

```
## [1] "2012-11-03"
```

```
dmy_hms("2/12/2013 14:05:01")
```

```
## [1] "2013-12-02 14:05:01 UTC"
```

```
mdy("120112")
```

```
## [1] "2012-12-01"
```

Handling Dates (continued)

- Not only the **lubridate** functions can be applied to vectors (returning a vector of results) but they are also very robust even to vectors containing dates in different formats:

```
dates <- c(20120521, "2010-12-12", "2007/01/5",
       "2015-2-04", "Measured on 2014-12-6",
       "2013-7+ 25")
dates <- ymd(dates) # produces a Date class
as.data.frame(dates)# converts to df for nicer display
```

```
##        dates
## 1 2012-05-21
## 2 2010-12-12
## 3 2007-01-05
## 4 2015-02-04
## 5 2014-12-06
## 6 2013-07-25
```

Handling Dates (continued)

- Frequently we also want to extract some of the temporal information on the dates and/or times we have received.

```
data.frame(Dates=dates,WeekDay=wday(dates),
      nWeekDay=wday(dates,label=TRUE),
      Year=year(dates),
      Month=month(dates,label=TRUE))
```

```
##         Dates WeekDay nWeekDay Year Month
## 1 2012-05-21       2      Mon 2012   May
## 2 2010-12-12       1      Sun 2010   Dec
## 3 2007-01-05       6      Fri 2007   Jan
## 4 2015-02-04       4      Wed 2015   Feb
## 5 2014-12-06       7      Sat 2014   Dec
## 6 2013-07-25       5      Thu 2013   Jul
```

Handling Dates (continued)

- **lubridate** can also help us to deal with different time zones. Dates/times may be measured in places under different time zones and we may wish to either keep that information or eventually convert between time zones.

```
date <- ymd_hms("20150823 18:00:05",
                tz="Europe/Berlin")
date
```

```
## [1] "2015-08-23 18:00:05 CEST"
```

- Converting this to the New Zealand time zone would give:

```
 with_tz(date,tz="Pacific/Auckland")
```

```
## [1] "2015-08-24 04:00:05 NZST"
```

- You can also change the time zone but maintain the same time:

```
force_tz(date,tz="Pacific/Auckland")
```

```
## [1] "2015-08-23 18:00:05 NZST"
```

Handling Dates (continued)

- The package **lubridate** also includes several other functions that help with handling time intervals or carrying out arithmetic operations with dates/times.
- Information on these features can be obtained in the respective package vignette.

Outliers

- Sometimes, data contain erroneous values resulting from measurement error, data entry error, or the like. If the erroneous value is in the same range as the rest of the data, it may be harmless. However, if it is well outside the range of the rest of the data, it may have a substantial effect on some of the data mining procedures.
- To detect outliers, analysts often use rules of thumb such as "anything over three standard deviations from the mean is an outlier."
- More formally, *outliers* are values that are far away from the bulk of the data.
- The term "far away" is deliberately left vague because what is or is not called an outlier (an erroneous or non-typical data entry) is an arbitrary decision that often requires a domain specific knowledge and judgement.

Outliers (continued)

- For example, a financial risk analyst may treat a once-in-a-century 70% decline in the stock marker as a valuable observation rather than an outlier.
- For small data sets, manual identification of outliers may be possible. However, for larger data sets, manual outlier detection is often not feasible so you must rely on data summarization and visualization methods. Some of those methods are the subject of our next module.
- One critical decision that must be made once an outlier is detected is how to deal with it. If the number of records with outliers is very small, they might be treated as missing data.

Outliers

- As a demonstration, consider the *FLOORS* variable in the West Roxbury housing data set.

```
table(housing.df$FLOORS)
```

```
##
##    1  1.5    2  2.5    3   15
## 1505  772 3415  105    4    1
```

- A simple tabulation of the the data reveals that there is one house with 15 floors. Since there is a little chance that a residential house has 15 floors, you should suspect that this observation is due to a misplaced decimal. If the suspicion is confirmed, then you should replace value 15 with 1.5.

```
tallbuilding.rowID<-rownames(
  housing.df[housing.df$FLOORS==15,])
# house in row 24 has 15 floors.
# Manually overwrite the number of floors to 1.5
housing.df[tallbuilding.rowID,"FLOORS"]<-1.5
```

Missing Values

- Often, the datasets you will be working on will contain records with missing values.
- If the number of records with missing values is relatively small, those records can simply be dropped from the data.
- As an alternative, consider simply replacing them with some imputed values, based on the other values for that variable across all the observations.
- E.g., if for some houses in our housing dataset the number of bedrooms is missing, we may consider substituting those with the mean (or median) number of bedrooms across all records. Doing so doesn't add any new information, it merely allows us to proceed with analysis and not lose the information contained in the existing data.

Missing Values (continued)

- To demonstrate the above procedure, let's first convert few entries in the housing data set into NA's, then impute these missing values using the median of the remaining values.

```
# randomy pick 10 rows
rows.to.missing<-sample(row.names(housing.df),10)
# substitute the bedroom data in the selected
# rows to NAs
housing.df[rows.to.missing,]$BEDROOMS<-NA
## get the summary. Use t(t()) for a nicer diplay
t(t(summary(housing.df$BEDROOMS)))
```

```
##          [,1]
## Min.     1.00
## 1st Qu.  3.00
## Median   3.00
## Mean     3.23
## 3rd Qu.  4.00
## Max.     9.00
## NA's    10.00
```

Missing Values (continued)

- The bedroom field now contains 10 missing values (NA's) and the median of the remaining values is 3.
- Now let's replace the missing values using the median of the remaining values.

```
housing.df[rows.to.missing,]$BEDROOMS<-
 median(housing.df$BEDROOMS, na.rm = TRUE)
# when computing the median, use na.rm=TRUE
# to ignore missing values.
t(t(summary(housing.df$BEDROOMS)))
```

```
##          [,1]
## Min.    1.000
## 1st Qu. 3.000
## Median  3.000
## Mean    3.229
## 3rd Qu. 4.000
## Max.    9.000
```

- No more missing values and the remaining descriptive statistics is the same.

Missing Values (continued)

- If the number of missing values is very large, dropping records/substituting them with imputed values will lead to a large loss of information. In such cases, analysts either utilize proxy variables with fewer missing values or methods or rely on the relationship between different predictors.
- E.g. if in the housing data set, the bedroom info is missing for many houses, consider using the lot size as a proxy for the bedrooms.
- Alternatively, consider first building a model that predicts the number of bedrooms using the houses with complete bedroom info, then use the model to input the predicted bedroom data for the houses where bedroom info missing.

Missing Values (continued)

- What if some values are missing but an identifier other than "NA" signifies a missing value. How do you deal with that?
- To demonstrate the problem, let's replace the bedroom values of the 10 selected rows with a "?"

```
# substitute the bedroom data in the selected
# rows to "?"s
housing.df<-housing.df
housing.df[rows.to.missing,]$BEDROOMS<-"?"
## get the summary. Use t(t()) for a nicer diplay
t(t(summary(housing.df$BEDROOMS)))
```

```
##        [,1]
## Length 5802
## Class  character
## Mode   character
```

- Since all rows of a column in a data frame must be of the same class, R automatically converted the BEDROOM column into the character class (same as "?").

Missing Values (continued)

- One easy way of dealing with the above (in this case self-inflicted) problem, is to resort to the facilities of the **readr** package once again

```
library(readr)
housing.df$BEDROOMS<-
  parse_integer(housing.df$BEDROOMS, na="?")
summary(housing.df$BEDROOMS)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.00    3.00    3.00    3.23    4.00    9.00      10
```

- Function `parse_integer()` can be used to parse a vector of character values into integers and to specify values that should be interpreted as NA's.
- Just for continuity, again replace NAs with the median

```
housing.df[rows.to.missing,]$BEDROOMS<-
 median(housing.df$BEDROOMS, na.rm = TRUE)
```

Missing Values (continued)

- Dealing with missing values could be a lengthy and painstaking process. In messy databases, for example, a "0" might mean two things: 1. the value is missing or 2. the value is actually zero. In such instances, there is no automated way of dealing with the problem.
- For example, you might have noticed that in the `YR_BUILT` variable, there is one record with value 0. Try running

```
housing.df[housing.df$'YR_BUILT'==0,]
```

- Since there is no chance that a house was built in year 0, we may proceed and substitute it with the median *YR_BUILT*.

```
missingyr.row<-rownames(
  housing.df[housing.df$'YR_BUILT'==0,])
missingyr.row # row with YR_BUILT = 0
```

```
## [1] "1493"
```

```
housing.df[missingyr.row,'YR_BUILT']<-
  median(housing.df$'YR_BUILT', na.rm=TRUE)
```

Missing Values (continued)

- R has several packages that provide tools to handle unknown values. Examples include functions on the **DMwR2** package, to more specific packages like **imputeR** (Feng et al., 2014)
- For more elaborated/accurate approaches to dealing with missing values refer to Torgo (**TO**, 2017, Section 4.5).

Normalizing (Standardizing) and Rescaling Data

- Numeric variables sometimes have rather different scales. This can create problems for some data analysis tools.
- To avoid these problems, we frequently apply some transformations to the original values of these numeric variables.
- Standardization is a frequently used technique that creates a new transformed variable with mean zero and unit standard deviation. It consists of applying the following formula to to the original values.

$$Y = \frac{X - \bar{x}}{S_X}$$

  where $\bar{x}$ is the sample mean of the original variable X, while S_X is its sample standard deviation.
- In R, function `scale()` performs this operation.
- To demonstrate, suppose we wanted to standardize the *LIVING_AREA* variable in our Roxbury Housing data set.

Normalizing (Standardizing) and Rescaling Data (cont.)

- To demonstrate, suppose we wanted to standardize the *LIVING_AREA* variable in our West Roxbury housing data set.

```
library(dplyr)
housing.df.livstan<-
  cbind(scale(select(housing.df,"LIVING_AREA")),
                select(housing.df,-"LIVING_AREA"))
summary(housing.df.livstan$"LIVING_AREA")
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -2.1335 -0.6459 -0.2009  0.0000  0.4009  6.7201
```

```
sd(housing.df.livstan$"LIVING_AREA")
```

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

- The resulting *LIVING_AREA* variable has a zero mean and unit standard deviation.
- Note that the above code used the *select()* function of the **dplyr** package, so that the resulting data frame is of the same class as the original.

Normalizing (Standardizing) and Rescaling Data (cont.)

- Another popular approach is rescaling certain variables to a $[0, 1]$ scale. This can achieved by subtracting from each observation the minimum value of the variable and dividing the difference by the range:

$$Y = \frac{X - \min X}{\max X - \min X}$$

In R, rescaling can be done using function `rescale()` of the **scales** package.

Normalizing (Standardizing) and Rescaling Data (cont.)

- Let's see how to apply the `rescale()` function to the *LIVING_AREA* variable in our West Roxbury housing database.

```
library(dplyr)
library(scales)
housing.df.rescaled<-
    cbind('LIVING_AREA'=
          rescale(housing.df$"LIVING_AREA"),
          select(housing.df,-"LIVING_AREA"))
summary(housing.df.rescaled$"LIVING_AREA")
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.1680  0.2183  0.2410  0.2863  1.0000
```

- As you can observe, rescaled *LIVING_AREA* ranges between 0 and 1.

Discretizing Variables

- Another frequently used data pre-processing technique is the discretization of numeric variables by transforming them into factors with meaningful bins.
- For instance, if we have a numeric column with the age of some clients, it may make sense for some applications to discretize these ages into "young", "adults", or "seniors", with the limits of these bins being determined by the background knowledge of the application (i.e. the end users).
- If no such domain knowledge exists one can use other criteria to determine the break points of the continuous variables domains. Two frequent choices are:
  (i) equal width that consists of splitting the range of the variable into k equal size bins; and
  (ii) equal frequency that makes sure every bin contains the same number of values appearing in the original dataset.

Discretizing Variables (continued)

- Whatever method we select, we can easily implement it through the functions `cut2()` of package **Hmisc** (Harrell Jr. et al., 2015) and `cut()` from base R.
- Suppose we wanted to create a new variable that denotes the current (2019) age of each housing unit in our housing data set:

```
library(Hmisc)
housing.df$AGE<-
  2019-housing.df$'YR_BUILT'
   t(t(summary(housing.df$'AGE')))
```

```
##          [,1]
## Min.       8.00
## 1st Qu.   64.00
## Median    84.00
## Mean      81.92
## 3rd Qu.   99.00
## Max.     221.00
```

Discretizing Variables (continued)

- If you want to apply the equal width discretization method to the AGE variable then it is simpler to use the cut() function by specifying the number of bins you want:

```
housing.df$'NEWAGE'<-cut(housing.df$AGE,3)
# counts of factors in each bin
table(housing.df$NEWAGE)
```

```
##
## (7.79,79]  (79,150] (150,221]
##      2618      3151        33
```

```
# To use own labels for the bins
housing.df$'NEWAGE'<-cut(housing.df$AGE,3,
           labels = c('new','normal','old'))
table(housing.df$NEWAGE)
```

```
##
##    new normal    old
##   2618   3151     33
```

Discretizing Variables (continued)

- If we want to apply the equal frequency method to this variable then function cut2() is more convenient.

```
library(Hmisc)
housing.df$'NEWAGE'<-cut2(housing.df$AGE,g=3)
table(housing.df$NEWAGE)
```

```
##
## [ 8, 70) [70, 95) [95,221]
##     2146     1953     1703
```

- Note from the output that although the bins above have more evenly distributed number of houses than the result of the `cut()` function, they are is still not exactly even.
- Since the cutting algorithm needs to put all houses with the exact same age into the same bucket, this leads to some imbalances in the buckets.

Discretizing Variables (continued)

- One drawback of the `cut2()` function is that it does not allow you to specify the labels of the bins which you need to do separately

```
housing.df$'NEWAGE'<-factor(cut2(housing.df$AGE,g=3),
           labels = c('new','normal','old'))
table(housing.df$NEWAGE)
```

```
##
##     new normal     old
##    2146   1953   1703
```

- Other more sophisticated discretization methods exist. Further information can be obtained for instance in Dougherty et al. (1995).