DATA TREATMENT WHITE PAPER

Preparing data for analysis using R

Nina Zumel, Win-Vector LLC

March 2016

INTRODUCTION

Cleaning and preparing data makes up a substantial portion of the time and effort spent in a data science project—the majority of the effort, in many cases. It can be tempting to shortcut this process and dive right into the modeling step without looking very hard at the data set first, especially when you have a lot of data. Resist the temptation. No data set is perfect; you will be missing data, have misinterpreted data, or have incorrect data. Some data fields will be dirty and inconsistent. If you don't take the time to examine the data before you start to model, you may find yourself redoing your work repeatedly as you discover bad data fields or variables that need to be transformed before modeling. In the worst case, you'll build a model that returns incorrect predictions—and you won't be sure why. By addressing data issues early, you can save yourself some unnecessary work, and a lot of headaches!

In this paper, we'll demonstrate some of the things that can go wrong with data, and explore ways to address those issues using the R statistical language **(https://cran.r-project.org/)** before going on to analysis. For faster numerical libraries, we will use the Microsoft R Open distribution **(https://mran.microsoft.com/open/)**. Throughout this discussion, we will keep an idealized goal in mind: using machine learning to build a predictive model.

A taxonomy of pain points

Several issues crop up again and again when preparing data for analysis in R:

- Loading data into R from databases, spreadsheets, or other formats
- Shaping data into a form appropriate for analysis
- Checking variable types: Are the variables of the type that you expect?
- Managing bad values: What do you do with NaNs or values that are out of range or invalid? How do you find and deal with sentinel values?
- Dealing with missing values (NA)
- Anticipating future novel categorical values (values that were not present in training data)
- Re-encoding categorical values with too many levels: How do you use such variables in an analysis efficiently?

Some of these problems must be dealt with in a domain-specific or problem-specific way. Other issues are more general and are good candidates for automated data treatment. We will touch on the first two issues briefly, and the remaining issues in more depth.

Loading data

There are a variety of functions and packages for loading data into R. Here, we touch on a few of our preferred functions and packages for loading data from the most common data sources.

Data source	Package	Function	Comments
Fixed-field text file (no delimiters)	utils	read.fwf()	Assumes fields are in fixed position on each line.
Delimited text file (comma-separated, tab-separated, etc.)	utils	read.table() reads data file into a data frame	Assumes data is in a regular, tabular format (no ragged rows).
Database	RJDBC	dbConnect() to establish connection dbReadTable() to read entire table into data frame dbGetQuery() to access data via SQL	Requires Java VM and JDBC driver. Supplies a good, concrete interface to many databases.
Excel spreadsheets	gdata	read.xls()	Requires perl. There are other packages, but gdata is the most reliable in our experience.
XML or JSON	XML or rjson		Fairly complicated; see package reference manuals.

Shaping data

The "shape" that is most efficient for recording or storing data information is not always the best shape for analyzing the data. For example, log data generally comes in a long and skinny format where information about a single entity (for example, a single customer or a single machine) is scattered across many rows.

```
##    id  fact       meas
## 1   1 fact1  2.2328720
## 2   2 fact1  4.3525637
## 3   3 fact1  4.2491274
## 4   4 fact1  2.3349491
## 5   1 fact2  1.4937386
## 6   2 fact2 -5.7002859
## 7   3 fact2 -1.0528365
## 8   4 fact2 -3.3691253
## 9   1 fact3  0.3009231
## 10  2 fact3  4.9687247
## 11  3 fact3 -6.0937292
## 12  4 fact3  5.9769071
```

For analysis, we generally prefer data in a wide format, where all facts about a single entity are stored in a single row.

```
##   id    fact1     fact2      fact3
## 1  1 2.232872  1.493739  0.3009231
## 2  2 4.352564 -5.700286  4.9687247
## 3  3 4.249127 -1.052836 -6.0937292
## 4  4 2.334949 -3.369125  5.9769071
```

This wide format is so central to R that R calls rows *observations* and columns *variables*.

R provides several functions for converting data from skinny to wide formats, and vice versa. For beginners, we recommend the **reshape2** package. To convert from skinny to wide data, use **dcast()**. To convert in the opposite direction, use **melt()**. More advanced users may wish to move on to the **dplyr** and **tidyr** ecosystems, though this involves learning additional notation.

Data munging and aggregation is a topic in itself (see **http://blog.revolutionanalytics.com/2014/01/fast-and-easy-data-munging-with-dplyr.html** for an introductory discussion). In this article, we will concentrate on data cleaning.

Variable types

Once you have your data loaded into R, it's time to check that all of it is as you expect. One of the advantages of data analysis in R is that R gives you many tools to explore and examine your data, and to fix the problems that you find. Checking that each variable is the type you expect seems trivial, but incorrect data types can mask some insidious data issues. For example, suppose we run **summary()** on our data and get this:[1]

```
##       age               education     marital.status
##  Min.   :17.00   HS-grad     :15784   Min.   :1.000
##  1st Qu.:28.00   Some-college:10878   1st Qu.:3.000
##  Median :37.00   Bachelors   : 8025   Median :3.000
##  Mean   :38.64   Masters     : 2657   Mean   :3.619
##  3rd Qu.:48.00   Assoc-voc   : 2061   3rd Qu.:5.000
##  Max.   :90.00   11th        : 1812   Max.   :7.000
##                  (Other)     : 7625
##            occupation     capital.gain    capital.loss   hours.per.week
##  Prof-specialty : 6172   Min.   :    0   0      :46560   Min.   : 1.00
##  Craft-repair   : 6112   1st Qu.:    0   1902   :  304   1st Qu.:40.00
##  Exec-managerial: 6086   Median :    0   1977   :  253   Median :40.00
##  Adm-clerical   : 5611   Mean   : 1079   1887   :  233   Mean   :40.42
##  Sales          : 5504   3rd Qu.:    0   2415   :   72   3rd Qu.:45.00
##  (Other)        :16548   Max.   :99999   1485   :   71   Max.   :99.00
##  NA's           : 2809                   (Other): 1349
##    income
##  large: 7841
##  small:24720
##  NA's :16281
##
##
##
##
```

Categorical variables masquerading as numbers

One of the first things we notice is that **marital.status** is a numeric value, which doesn't make sense. This can be an issue for some machine learning algorithms (like regression), which will try to treat this variable as if it is a continuous numerical value. This can lead to incorrect inferences.

Fortunately, R, unlike most programming languages, has a concept for representing categorical variables: the *factor* class. The most straightforward solution is simply to convert the numeric column to a factor column.

```
# check that marital.status is probably a code
unique(adultf$marital.status)
```

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

```
# easiest solution: just change the column to a factor
newvar = as.factor(adultf$marital.status)
summary(newvar)
```

```
##     1     2     3     4     5     6     7
##  6633    37 22379   628 16117  1530  1518
```

[1] This example data set is derived from the AdultUCI data set in the package arules, with some modifications to demonstrate various issues.

If you have access to the data dictionary, you can convert the numeric codes to more meaningful strings—another advantage R has over other programming languages that are used for analysis.

```
#
# better solution: use data dictionary to map integer code to meaningful strings
#
# load data dictionary
marital_map = readRDS("maritaldict.rds")
# marital_map is a vector of category level descriptions, indexed (and named) by the category code
marital_map
```

```
##                        1                        2                        3
##               "Divorced"     "Married-AF-spouse"    "Married-civ-spouse"
##                        4                        5                        6
## "Married-spouse-absent"         "Never-married"              "Separated"
##                        7
##                "Widowed"
```

```
newvar = marital_map[adultf$marital.status]
head(newvar)
```

```
##                    5                    3                    1
##      "Never-married" "Married-civ-spouse"           "Divorced"
##                    3                    3                    3
## "Married-civ-spouse" "Married-civ-spouse" "Married-civ-spouse"
```

```
# now marital.status is a category variable (factor) with meaningful level names
adultf$marital.status = as.factor(newvar)
summary(adultf$marital.status)
```

```
##              Divorced     Married-AF-spouse    Married-civ-spouse
##                  6633                    37                 22379
## Married-spouse-absent         Never-married             Separated
##                   628                 16117                  1530
##               Widowed
##                  1518
```

The above code takes advantage of the fact that in R elements of a vector can be accessed by name as well as by position. We can use this ability to build a named vector that maps the numeric codes (or more exactly, the string representation of them) to more meaningful names for the category levels. In R this remapping fixed many rows all at once; a simple for-loop over column names can repair many columns.

Strings or factors where you expect numbers

Now let's look at the variables **capital.gain** and **capital.loss**. We would expect that **capital.gain** and **capital.loss** should both be numeric monetary values. However, we can see that **capital.loss** is not a numeric variable; it is a factor (it may also be a string, if the data were read into R using **read.table()** with the parameter **stringsAsFactors=FALSE**). Most likely, nonnumeric characters caused R to read in this column as character valued (strings) rather than as numbers. We can check this.

```
# convert the strings in the column to numeric
closs = as.numeric(as.character(adultf$capital.loss))
```

```
## Warning: NAs introduced by coercion
```

```
# check the values that did not convert
adultf$capital.loss[is.na(closs)]
```

```
## [1] 1,579 2,559 1,504 1,876 2,002
## 104 Levels: 0 1,504 1,579 1,876 1092 1138 1258 1340 1380 1408 1411 ... 974
```

In this case, we see that commas in a few of the entries are causing the problem. String substitution to remove the commas and then as.numeric to convert the column will fix this.

```
# use gsub to remove the commas from the data
closs_str = as.character(adultf$capital.loss)
closs = gsub(",", '', closs_str, fixed=TRUE)
adultf$capital.loss = as.numeric(closs)
summary(adultf$capital.loss)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     0.0     0.0    87.5     0.0  4356.0
```

An especially subtle issue is when the corrupted variable is a numeric identifier column (like customer id or social security number). You may have several tables, all of which are indexed by this identifier column, and you might want to merge data from these disparate tables into a single table using (for example) **customer_id** as the merge key. If the **customer_id** column has been corrupted in one of the tables, and was read in as a string or factor variable rather than as a numeric variable, your merge will fail, possibly without you noticing.

Check for bad or missing values

Bad values can be missing **(NA)** or problematic types **(NaN, Inf)**. They can also be invalid values: invalid category levels, implausible numeric values (negative ages; values that are outside the range a variable would be expected to take). Identifying bad values often requires domain knowledge of the plausible values of the variables.

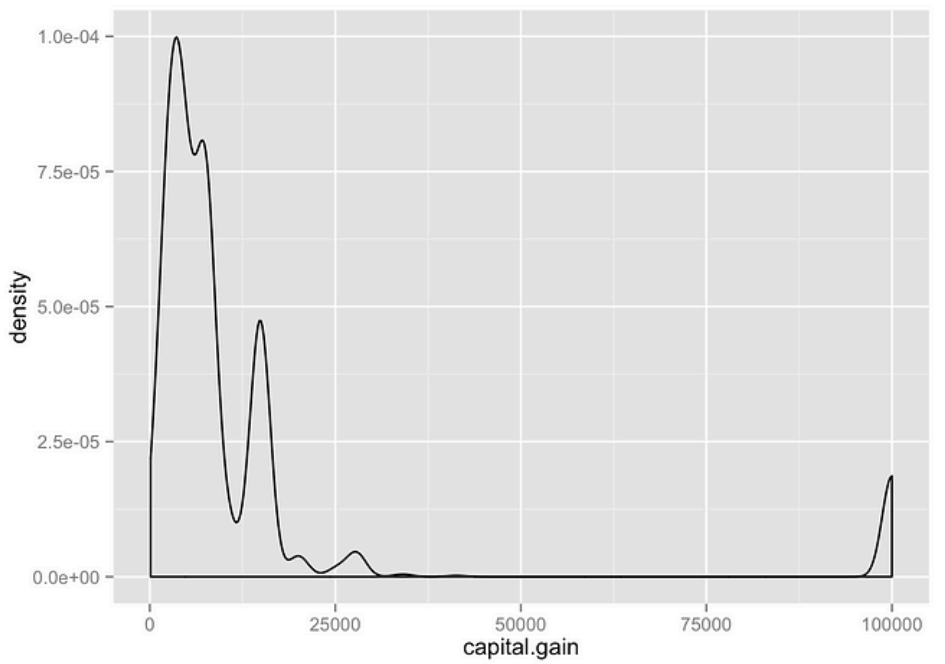
A special kind of bad numerical value is the *sentinel value:* a value that used to represent "unknown" or "not applicable" or other special cases in numeric data. A **-1** in age data can be a sentinel value, meaning "age unknown." All 9s is also a common sentinel value. Sometimes, the maximum value of a variable may really be the maximum recorded value; any value greater than that maximum was censored down.

Detecting sentinel values

One way to detect sentinel values is to look for sudden jumps in an otherwise smooth distribution of values. In the **adultf** data, the **99999** value for **capital.gain** looks suspiciously like a sentinel. We can check for that by graphing the distribution of the **capital.gain** variable.

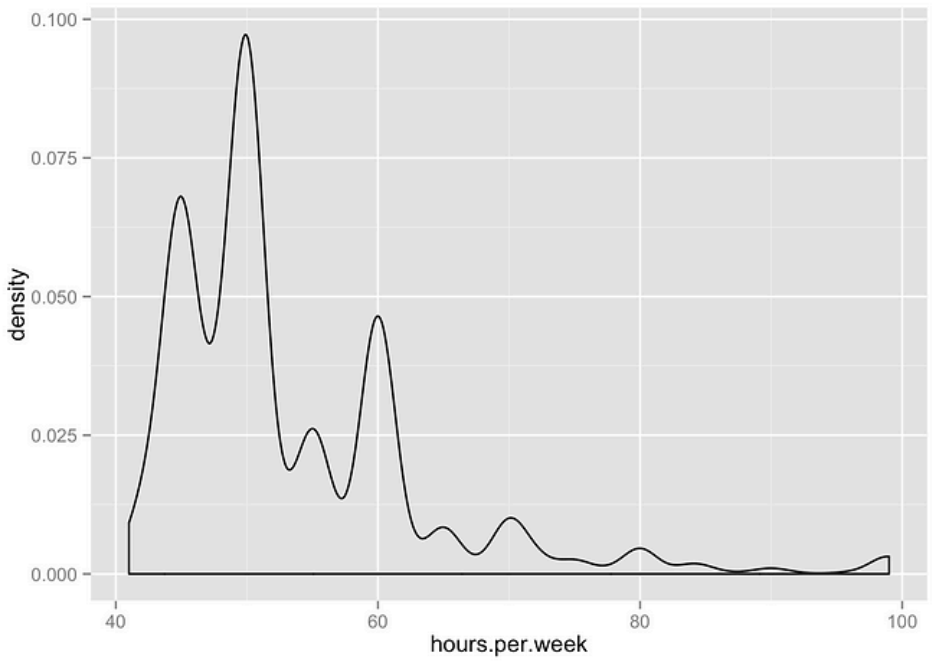
```
library(ggplot2)

# look at the distribution of capital.gain. Ignore the spike at 0
ggplot(subset(adultf, capital.gain>0), aes(x=capital.gain)) + geom_density()
```

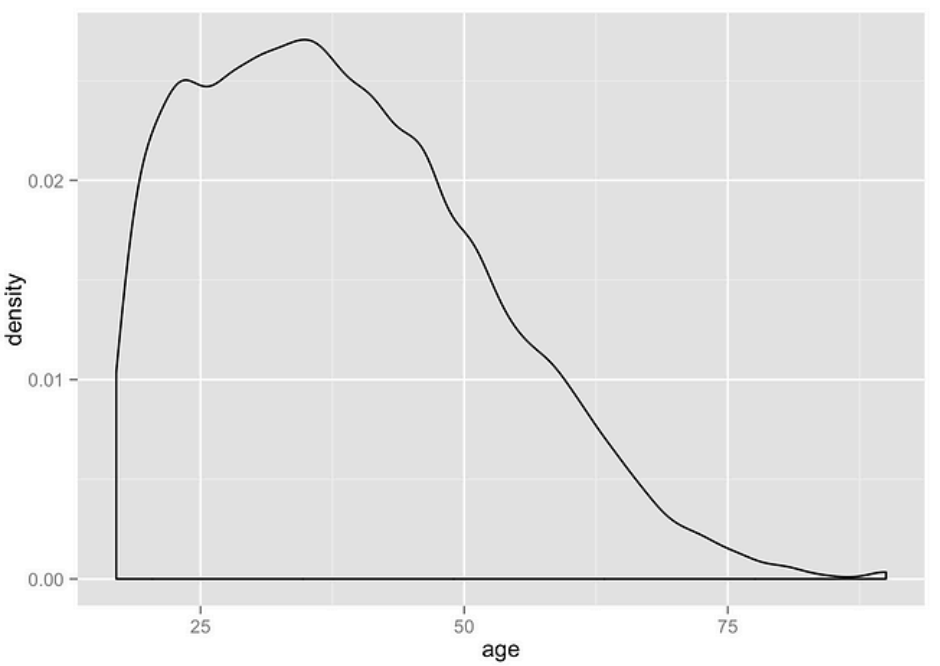


The sudden uptick in the distribution at 99999 is good evidence that it is a sentinel value for **NA** or "unknown." We can see similar suspicious spikes at the 99 of the **hours.per.week** variable and the 90 of the **age** variable. These "sneaky sentinel values" arise because many systems lack a uniform notation for "no value."

```
# look at the distribution of hours.per.week. Ignore the spike at 40
ggplot(subset(adultf, hours.per.week>40), aes(x=hours.per.week)) + geom_density()
```




```
# look at the distribution of age
ggplot(adultf, aes(x=age)) + geom_density()
```



The slight uptick in each distribution is mild evidence that **age** was capped at 90 (so 90 really means "90 or older") and **hours.per.week** was capped at 99 (so 99 really means "99 hours or more"). In these two cases, the uptick is not that dramatic, and the number of problematic cases is small, so it may be safe to treat the data as given. However, in the **capital.gain** case, the sentinel value is dramatically out of range with the rest of the data. To deal with it, we will first convert the sentinel values to **NA**, so that we don't mistake them for actual capital gains.

```
sentinel = 99999
isbad = which(adultf$capital.gain==sentinel)
length(isbad) # the number of bad values
```

```
## [1] 244
```

```
adultf$capital.gain[isbad] = NA
summary(adultf$capital.gain)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     0.0     0.0     0.0   582.4     0.0 41310.0     244
```

For more on graphing, exploring, and repairing data see Chapters 3 and 4 of *Practical Data Science with R* (Zumel and Mount, Manning 2014).

Dealing with missing values (NA)

To drop or not to drop?

If the number of missing values is small, it may be safe to simply drop those rows. However, take care: a few missing values in this variable and a few missing values in that variable can quickly add up to a lot of incomplete data. The function **complete.cases()** on a data frame returns **TRUE** for every row where there are no **NA**s, **FALSE** otherwise.

```
nrows = nrow(adultf) # how many rows of data
ncomplete = sum(complete.cases(adultf)) # how many complete rows of data
ncomplete/nrows
```

```
## [1] 0.6257524
```

In this case, dropping all the rows with missing values eliminates almost 40 percent of the data. This can lead to incorrect models, especially when there are additional distributional differences between the dropped and retained data. Also, production models usually need to score all observations, even when those observations have missing values.

Categorical variables

NAs in categorical variables can be treated as an additional category level.

```
summary(adultf$occupation)
```

```
##      Adm-clerical      Armed-Forces      Craft-repair   Exec-managerial
##              5611                15              6112              6086
##   Farming-fishing Handlers-cleaners Machine-op-inspct     Other-service
##              1490              2072              3022              4923
##   Priv-house-serv    Prof-specialty   Protective-serv             Sales
##               242              6172               983              5504
##      Tech-support  Transport-moving              NA's
##              1446              2355              2809
```

```
# get the strings corresponding to the factor levels
occstr = as.character(adultf$occupation)

# Convert NA to "unknown." Although technically we don't know if NA properly means
# "unknown" or "no occupation."
occstr = ifelse(is.na(occstr), "unknown", occstr)
adultf$occupation = as.factor(occstr)
summary(adultf$occupation)
```

```
##      Adm-clerical      Armed-Forces      Craft-repair   Exec-managerial
##              5611                15              6112              6086
##   Farming-fishing Handlers-cleaners Machine-op-inspct     Other-service
##              1490              2072              3022              4923
##   Priv-house-serv    Prof-specialty   Protective-serv             Sales
##               242              6172               983              5504
##      Tech-support  Transport-moving           unknown
##              1446              2355              2809
```

Numerical variables

Treating **NA**s in numerical variables depends on why the data is missing.

When values are missing randomly

Let's consider the **capital.gains** variable, which now has several missing values. You might believe that the data is missing because of a "faulty sensor"—in other words, the data collection failed at random (independently of the value sensored, and all other variables or outcomes). In this case, you can replace the missing values with stand-ins, such as inferred values, distributions of values, or the expected or mean value of the nonmissing data. Assuming that the customers with missing capital gains are distributed the same way as the others, this estimate will be correct on average, and you'll be about as likely to have overestimated capital gains as underestimated it. It's also an easy fix to implement.

This estimate can be improved when you believe that capital gains is related to other variables in your data—for instance, income or occupation. If you have this information, you can use it. Note that the method of imputing a missing value of an input variable based on the other input variables can be applied to categorical data as well. The text *R in Action, Second Edition* (Robert Kabacoff, Manning 2015) includes an extensive discussion of several methods for imputing missing values that are available in R.

When values are missing systematically

It's important to remember that replacing missing values by the mean, as well as many more sophisticated methods for imputing missing values, assumes that the rows with missing data are in some sense random (the "faulty sensor" situation). It's possible that the rows with missing data are systematically different from the others. For example, if we were looking at data about cars, then cars with missing fuel efficiency (miles per gallon) data might be electric cars, for which the concept of "miles per gallon" is meaningless. In that case, "filling in" the missing values using one of the preceeding methods is the wrong thing to do. You don't want to throw out the data either. In this situation, a good solution is to fill in the missing values with a nominal value, either the mean value of the nonmissing data or zero, and additionally to add a new variable that tracks which data have been altered:

```
summary(adultf$capital.gain)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##     0.0     0.0     0.0   582.4     0.0 41310.0     244
```

```
capgain = adultf$capital.gain
meancg = mean(capgain, na.rm=TRUE)
meancg
```

```
## [1] 582.4121
```

```
cg.isbad = is.na(capgain)
capgain = ifelse(is.na(capgain), meancg, capgain)
summary(capgain)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     0.0     0.0     0.0   582.4     0.0 41310.0
```

```
summary(cg.isbad)
```

```
##    Mode   FALSE    TRUE    NA's
## logical   48598     244       0
```

```
adultf$capital.gain = capgain
adultf$capital.gain.isbad = cg.isbad
```

If you don't know for sure whether missing values in your data are missing randomly or systematically, it is safer and more conservative to assume that they are missing systematically. For business data, this is the more likely scenario. And often (because missingness is often an indication of data provenance) the **isbad** column can be a highly informative variable—sometimes more informative than the actual values that the variable takes on.

Categorical variables with too many levels or with rare levels

Another type of problematic variable is the categorical variable with many possible values (or *levels,* in R parlance): zip codes and business codes like NAICS codes fall into this category. Computationally speaking, a categorical variable with *k* levels is treated by most machine learning algorithms as the equivalent of about *k* numerical (0/1) variables. There are roughly 40,000 zip codes in the United States—far too many variables for most machine learning algorithms to handle well.

In addition, if a categorical variable can take on many levels relative to the size of the data, then most of those levels will be rare, and hence more difficult to learn. It is also more likely that many allowable levels won't even show up in the training data, which in turn causes problems if these unseen levels appear when you are applying your model to new data. Of course, this *novel level* problem is potentially a problem even with categorical levels with fewer levels, if some of those levels are rare.

Dealing with novel levels

Most of R's machine learning algorithm implentations will crash when they try to apply a model to data that contains a level they don't recognize (that is, a level that didn't appear in the training data). To avoid this, you can preprocess your data before presenting it to your model to detect these novel levels and convert them to something that the model can deal with. For example, if before training you pooled all rare levels in the training data (into a single level called **rare** for example), then you can also map novel levels in new data to **rare** before applying the model.

Dealing with too many levels

If your data has a variable like zip code that has too many levels for a machine learning algorithm to deal with, you can try converting that variable to a numerical variable (or to a categorical variable with fewer categories) in one of two possible ways:

Look-up codes
Often a variable like zip code or NAICS code is really a proxy for demographic or other information, and it's this other information that's really of interest. For example, if you are trying to build a model to predict someone's income or their net worth, then the average or median income of people in a certain zip code is useful information. If you have access to external information about average income by zip code, then you can use the zip codes in your data to look up those average incomes, and use that data in your model.

One-variable models (impact coding)
Alternately, you can convert the problematic variable into a single-variable submodel for the desired income; essentially, a lookup table or a conditional probability model based on the training data. For example, if you are interested in predicting income, then you can build a submodel of income by zip code by finding the average income per zip code in a random subset of your training data. If you are building a classifier (perhaps building a model to predict "high income," for some definition of "high"), then you can use part of the training data to build a submodel that predicts the conditional probability of high income, conditioned on zip code.

Once you have a submodel, then you can use the predictions of that submodel as a variable in your overall model in place of the original problematic variable. This technique is called *effects modeling* in some disciplines, and we refer to it as *impact modeling* in this blog post: **http://www.win-vector.com/blog/2012/07/modeling-trick-impact-coding-of-categorical-variables-with-many-levels/** which discusses the technique in more detail.

Here we show a simple example of impact modeling where the outcome to be predicted is income class (**large** or **small**), and the variable to be impact-coded is education level.

```
# build a simple impact model of income conditioned on education,
# based on the data set cal
impact_table = table(cal[, c("education", "income")])
edutab = rowSums(impact_table)
impact_table = impact_table/edutab
pLargeIncome_education = impact_table[,"large"]

# pLargeIncome_education now returns the conditional probability of high income (income="large"),
# given education level. In practice, we should add some smoothing to avoid probability values of 0 or 1
pLargeIncome_education
```

```
##         10th         11th         12th      1st-4th      5th-6th
##   0.06153846   0.06190476   0.05797101   0.00000000   0.04838710
##      7th-8th          9th    Assoc-acdm     Assoc-voc    Bachelors
##   0.04597701   0.02702703   0.26900585   0.27188940   0.43536585
##    Doctorate      HS-grad      Masters    Preschool  Prof-school
##   0.73770492   0.16687737   0.52713178   0.00000000   0.69863014
## Some-college
##   0.21383648
```

```
# apply the impact model to the education levels in the training set
head(train$education)
```

```
## [1] Bachelors 11th      Bachelors 9th       HS-grad   Masters
## 16 Levels: 10th 11th 12th 1st-4th 5th-6th 7th-8th 9th ... Some-college
```

```
education_impact = pLargeIncome_education[as.character(train$education)]

# the numerical variable education_impact can now be used in a larger model for income
head(education_impact)
```

```
##  Bachelors       11th  Bachelors        9th    HS-grad    Masters
## 0.43536585 0.06190476 0.43536585 0.02702703 0.16687737 0.52713178
```

Impact coding in the **vtreat** package

Impact coding is implemented in the R package **vtreat**, and here we show a small example of its use.

```
# for simplicity, just consider education
set.seed(34597)

incomeByEdu = adultf[!is.na(adultf$income), c("education", "income")]
gp = runif(nrow(incomeByEdu))

# split data training-cal-test 70-15-15
train = incomeByEdu[gp < 0.7,]
cal = incomeByEdu[0.7 <= gp & gp < 0.85,]
test = incomeByEdu[gp >= 0.85,]

library(vtreat)

# create treatment plan. "large" is the positive income class
treatplan = designTreatmentsC(cal, "education", "income", "large")
```

```
# apply the treatment plan to the training set
train.treat = prepare(treatplan, train, pruneSig=NULL)

# the values from the Bayesian impact model are now in the variable education_catB
# the values are log(p(posClass | category)/p(posClass))
summary(train.treat$education_catB)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -10.34000  -0.39510  -0.14710  -0.25120   0.09304   1.09100
```

The function **designTreatmentsC()** creates what we call a treatment plan: a record of all the summary statistics needed to create and apply the single-variable impact model for predicting a categorical outcome. We apply the treatment plan to the training data to get back the predictions of the impact model (in **train.treat$education_catB**). The impact model in **vtreat** returns the log ratio of the conditional probability of high income given education level to the overall probability of high income. In other words, it encodes whether or not a given education level has a positive or negative impact on the probability of high income, compared to the average.

The predictions from the impact model can then be used in a model for income, along with any other variables like age or occupation.

In order to apply the resulting model to new data (in this case, the data set **test**), we again apply the treatment plan to **test** to get the predictions of the impact model on **test$education**. These predictions, in **test.treat$education_catB**, can then be fed to the overall income model.

```
train.treat = prepare(treatplan, train, pruneSig=NULL)

summary(train.treat$education_catB)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -10.34000  -0.39510  -0.14710  -0.25120   0.09304   1.09100
```

The **vtreat** impact model also handles novel levels and missing data. In fact, **vtreat** creates general treatment plans that can also manage missing data for categorical variables in general (that is, without creating impact models), and handles bad and missing values in numerical data as well. However, a general discussion of **vtreat** is outside the scope of this paper.

Note that we used different data sets to create the treatment plan/impact model (the **cal** data set) and to train the overall income model (the **train** data set). This is recommended practice, because using the same data to create the impact model and the larger overall model can lead to undesirable bias in model training.

CONCLUSION

In this paper, we've looked at a number of common pain points that arise when preparing your data for analysis. We've shown how to detect or anticipate these issues, and how to address them using R. Some of these tasks can be readily automated, but some will be more domain specific and must be handled on a case-by-case basis.