Encoding Categorical Predictors

Agazzi Ruben Davide Dell'Orto Hellem Carrasco

Abstract

This is a comprehensive analysis of how categorical, and more general non numerical data can be encoded into a numerical form, in order to be used by all types of models. In particular will be addressed how to manage unordered and ordered categorical data, and also how to extract features from textual data.

Summary

Chapter 1

Handling categorial data is a very important step during the implementation of a statistical or machine learning model. Most of the models does not accept categorical data, but only numerical data. The only models that accepts this type of data are tree-based model which can handle this type of data by default.

Dummy Variables

The most basic approach used for handling unordered categorical data, consists in creating dummy or indicator variables. The most common form is creating binary dummy variables for categorical predictors: if the categorical predictor can assume 3 values we can create 2 binary dummy variables where value 1 will be, for example, d1=1 and d2=0, value 2 will be d1=0 and d2=1 and where value 3 will be d1=0 and d2=0. We could also represent this type of categorical data with 3 dummy variables, but this approach could lead, sometimes, to problems: some models, in order to estimate their parameters, needs to invert the matrix $(X'X)$, if the model has an intercept an additional column of one for all columns is added to the X matrix; if we add the new 3 dummy variables this will add to the previous intercept row, and this linear combination would prevent the $(X'X)$ matrix to be invertible. This is the reason why we encode C different categories into $C-1$ dummy variables

Code

This piece of code shows a way of encoding the seven days of weeks into 6 dummy variables

```
library(tidymodels)
```

```
## -- Attaching packages --------------------------------------- tidymodels 1.0.0 --
```

```
## v broom        1.0.4     v recipes      1.0.5
## v dials        1.2.0     v rsample      1.1.1
## v dplyr        1.1.1     v tibble       3.2.1
## v ggplot2      3.4.2     v tidyr        1.3.0
## v infer        1.0.4     v tune         1.1.0
## v modeldata    1.1.0     v workflows    1.1.3
## v parsnip      1.0.4     v workflowsets 1.0.0
## v purrr        1.0.1     v yardstick    1.1.0
```

```
## -- Conflicts ------------------------------------------ tidymodels_conflicts() --
## x purrr::discard() masks scales::discard()
## x dplyr::filter()  masks stats::filter()
## x dplyr::lag()     masks stats::lag()
## x recipes::step()  masks stats::step()
## * Learn how to get started at https://www.tidymodels.org/start/
```

```
library(FeatureHashing)
library(stringr)
```

```
##
## Attaching package: 'stringr'
```

```
## The following object is masked from 'package:recipes':
##
##     fixed
```

```
library('fastDummies')
days_of_week <- c("Sunday", "Monday", "Tuesday", "Wednesday", "Thursday","Friday", "Saturday")
test_df <- data.frame(days_of_week)
test_df <- dummy_cols(test_df, select_columns = c('days_of_week'))
test_df = test_df[-c(5)]
```

days_of_week <chr>	days_of_week_Friday <int>	days_of_week_Monday <int>	days_of_week_Saturday <int>	days_of_week_Thursday <int>	days_of_week_Tuesday <int>	days_of_week_Wednesday <int>
Sunday	0	0	0	0	0	0
Monday	0	1	0	0	0	0
Tuesday	0	0	0	0	1	0
Wednesday	0	0	0	0	0	1
Thursday	0	0	0	1	0	0
Friday	1	0	0	0	0	0
Saturday	0	0	1	0	0	0

7 rows

Figure 1: Table of encoded days of week using 6 dummy variables

Encoding predictors mith many categories

We may have a problem by encoding values using simple dummy variables: if the categorical predictor can assume a large number of different values, like for example the ZIP code, we will end up with too many dummy variables. If we make a resampling on this type of data there is also the problem where some dummy variables will assume an always 0 value: this case is called *zero-variance predictor*. Using the example of ZIP codes, highli popolated areas will lead to higher frequencies of the same ZIP codes, while some will be very rare, in this case the dummy variables could become *near-zero-variance predictors*. In this cases we could remove *zero-variance and near-zero-variance predictors*.

Hashing functions

Instead of removing the mentioned types of predictors we could use an hashing function to map the values into fewer values, and then use these values to create dummy variables. the process of using hashes to create dummy variables is called *feature hasing* or *hash trick*. The process of hashing features is:

1. Calculate the hash from the feature value, which is an integer value.
2. Calculate the new feature value, if we want 16 different values we calculate *(hashValue mod 16) +1*

Some considerations needs to be done: because of the nature of the hashing functions, sometimes we will have collisions. Collisions occurs when two different values has different hashing. In statistics this is an aliasing problem. A way to solve this problem consists into using signed hashes, so instead of only 0 and 1 values for dummy variables, we can also have -1 value in order to reduce aliasing.

Problems of feature hashing

The process of feature hashing can lead to some problems:

1. Collisions: Even with signed hashes collisions can appear, and in this cases is more difficult to understand the impact of a category on the model because of this aliasing problem.
2. Collision meaning: if two categories collides, this does not mean that they share something in common or have some type of correlation or similarity.

3. Collision probability: as hashing function does not have information about the distribution of data, rarer categories can collide with more high frequency categories, in this case the higher frequency category will have much more influence.

Code

This piece of code shows how feature hashing can be done. The dataset used is the OkCupid dataset and the categorical predictor used was the *where_town* column of the dataset.

```
library(tidymodels)
library(FeatureHashing)
library(stringr)

options(width = 150)


load("./Datasets/okc.RData")

towns_to_sample <- c(
  'alameda', 'belmont', 'benicia', 'berkeley', 'castro_valley', 'daly_city',
  'emeryville', 'fairfax', 'martinez', 'menlo_park', 'mountain_view', 'oakland',
  'other', 'palo_alto', 'san_francisco', 'san_leandro', 'san_mateo',
  'san_rafael', 'south_san_francisco', 'walnut_creek'
)

# Sampled locations from "where_town" column
locations_sampled <- okc_train %>% dplyr::select(where_town) %>% distinct(where_town) %>% arrange(where_

hashes <- hashed.model.matrix(
  ~ where_town,
  data = locations_sampled,
  hash.size = 2^4,
  signed.hash=FALSE,
  create.mapping=TRUE
)

hash_mapping = hash.mapping(hashes)
names(hash_mapping) = str_remove(names(hash_mapping), 'where_town')

# Takes hash mapping, converts to dataframe, set new columns names, calculate hash over name to have or
binary_calcs = hash_mapping %>% enframe() %>% set_names(c('town', 'column_num_16')) %>% mutate(integer_

hashes_df = hashes %>%
  as.matrix() %>%
  as_tibble() %>%
  bind_cols(locations_sampled) %>%
  dplyr::rename(town = where_town) %>%
  dplyr::filter(town %in% towns_to_sample) %>%
  arrange(town)


# Making a signed hasing version in order to prevent aliasing

hashes_signed <- hashed.model.matrix(
  ~ where_town,
```

```
  data = locations_sampled,
  hash.size = 2^4,
  signed.hash=TRUE,
  create.mapping=TRUE
)
hashes_df_signed = hashes_signed %>%
  as.matrix() %>%
  as_tibble() %>%
  bind_cols(locations_sampled) %>%
  dplyr::rename(town = where_town) %>%
  dplyr::filter(town %in% towns_to_sample) %>%
  arrange(town)
```