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()
## * Search for functions across packages at https://www.tidymodels.org/find/
```

```
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)
```

Encoding of ordered data

Some categorical data can be ordered. One example of ordered categorical data, can be a review with values that are like "bad", "normal" and "good". In this case "bad" should have a different meaning with respect to the "good" case during the training of a model, and we should not only indicate the presence of the value "good" or "bad".

How to encode ordered categorical data

Ordered categorical data can have a different type of relationship between the values: for example the values can have a linear relationship or a quadratic relationship. We must encode these data with the right relationship in order to let the model understand this type of relationship. The encoding used to maintain this type of relationship is called *Polynomial Contrast*. A contrast has the characteristic that is a single comparison(one degree of freedom) and the sum of the coefficients is equal to zero. Polynomial contrast is useful, because it can represent linear relations, but also non linear shapes too: for example we can make a quadratic polynomial contrast, cubic polynomial contrast, etc. Polynomial constrast is also useful because it can be done on data with any number of ordered factors; the only "drawback" is that the complexity of the contrast can be, at most, equal to the number of categories of the predictor minus one: for example, if our ordered categorical predictor have three different levels, we can make only a linear and quadratic polynomial contrast. We can use *polynomial contrast* to investigate multiple relationships at the same time by including them in the same model, for example both linear and quadratic polynomial contrast.

Drawbacks

A drawback of the polynomial contrast is that it may not relate directly the predictor to the response: for example, in the case where two levels of the categorical ordered predictor are very close, like in the case of "low" and "middle", polynomial contrast does not encode well this situation and does not have particular effective response in modeling the predictor's trend. Another drawback happens when we have a high number of different C categories: Because we can make at most $C - 1$ degree polynomial contrast, if the number of C categories is very high, we can have higher grade contrasts, but in practice it has be seen that polynomial contrast is usually useful up to the quadratic polynomial contrast; in this case we might want to limit the maximum polynomial degree of the contrasts.

Alternatives

Some alternatives to polynomial contrast are:

1. Treat the predictors as unordered factors. This can be useful if the pattern of the categorical data is not polynomial, but if the underlying patter is linear or quadratic, unordered dummy variables may not uncover this trend.
2. Manually translate the categories into a set of numberic score, using domani-specific knowledge of the data.

Code

The following code illustrates how to obtain *polynomial contrast* encoding for an ordered categorical predictor with 3 number of categories.

```
# Linear and quadratic polynomial contrasts are generated, the scores -1,0,1 can be seen as "bad", "med
zapsmall(contr.poly(3, scores=c(-1,0,1)))
```

```
##               .L         .Q
## [1,] -0.7071068  0.4082483
## [2,]  0.0000000 -0.8164966
## [3,]  0.7071068  0.4082483
```

As we can see from the results of the code the linear *polynomial contrast* have as values -0.71, 0, 0.71 and for quadratic *polynomial contrast* value have 0.41, -0.82, 0,41. As we can see the two set of values has sum equal to 0.

Features from text data

Usually we can come around some textual data inside a dataset, for example a product description or a profile description. We need to find a way to extract some features from texts. One example can be searching for the presence of links inside the text. Like suggested in the book, we can make a new feature that indicates the presence of a link inside the profile description. This feature can be useful to be used to classify a profile, in order to predict if the profile belongs to a STEM or non-STEM person. With our testing dataset we found out that the profiles with a link in the description are 662, where 409 are STEM profiles and 253 are non stem profiles. The odds ratio of having a link in the STEM profiles is equal to 0.043, while the odds ratio of non-STEM profiles that have a link in the description is 0.026. This values makes us understand that the stem profiles that have a link in the description are 1.65 times higher than the non-STEM profiles with a link. To see if this proportion is reliable we made a confidence interval over the odds-ratio of profiles with link. The confidence interval at 95% have as lower bound 1.42 and as upper bound 1.96, in this case 1 is not included inside the interval, so we can say that this feature is not working only because of some random noise, but instead is a useful feature to distinguish profiles between STEM and non-STEM

Other features

By following the previous analysis, other features can be obtained, like the presence of certain keywords, or the count itself of a specified word, the number of commas, hashtags, exclamation points, etc.. In this case it is useful to remember that some form of preprocessing should be done when working with textual data. The preprocessing steps can consists in:

1. Stop-word removal.
2. Removal of undesired elements, like html tags and other useless parts for the feature extraction.
3. Lemmatization or stemming in order to reduce the words to the same form, in order to accorpate the terms
4. Making al the text lowercase to normalize it.
5. Remove words that have a very low frequency.

Analysis on text features

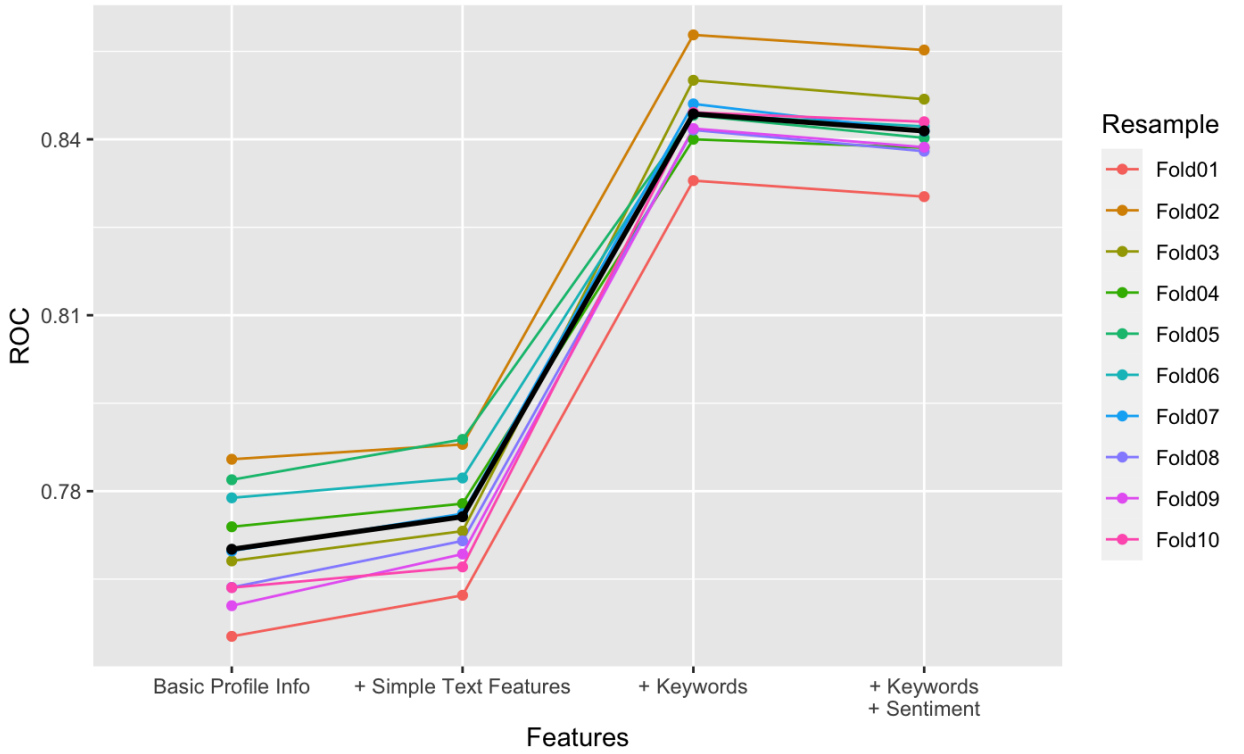
After the initial consideration about the presence of link feature and more in general of other features, we proceeded to make an analysis about the effectiveness of these features. To do so we used the different datasets, with different features inside, in order to train 4 different general linear models, in order to see how the performance is affected by the presence of different features. The 4 datasets sused for the training are:

1. Basic Ok Cupid dataset, with profile info consisting in 160 predictors, after creating dummy variables when needed.

2. Basic Ok Cupid dataset and text features like the number of occurences of urls, commas, exclamation points, etc.
3. Basic Ok Cupid dataset, text features and keyword features, like the presence of the "software" term in the description.
4. Basic Ok Cupid dataset, text features, keyword features and sentiment features, like the number of sentences, number of sentences written in first or third person, etc.

Performance evaluation

In order to evaluate the performances of the 4 different models, we used the area under the ROC curve. The training also performs 10-Fold cross validation in order to have more precise results. In the end the performance is calculated as the average of the 10 folds performances.



As we can see from the plot showing the performances, adding the simple text features to the dataset improves the area under the ROC courve by a little, while adding the keywords features greatly improves the performances. Adding the sentiment analysis features, on the other hand, decreases the performance with respect to the model trained with only the simple text features and keywords features.

Other considerations

Another way of extracting text features can be the implementation of tf-idf statistics to identify important words: this statistic measures the grade of importance of a word inside its context, so if is more present locally it is considered more important, and inside the global context of documents, so if a word is rare among all documents its importance is considered bigger. The good thing of tf-idf is that it can consider not only single words, but n-grams, and this could lead to find more useful features inside our textual data.

Code

The following code contains the analysis made on the odds ratio of the presence of link feature and the training and comparison of the 4 models with different feature sets.

```r
library(epitools)
load("./Datasets/okc.RData")

stem_data = okc_sampled[okc_sampled$Class == "stem",]
non_stem_data = okc_sampled[okc_sampled$Class != "stem",]
result_stem = grepl("http|https", stem_data$essays)
result_non_stem = grepl("http|https", non_stem_data$essays)

number_stem_with_link = sum(result_stem)
number_non_stem_with_link = sum(result_non_stem)


odds_stem = (number_stem_with_link ) / nrow(okc_sampled)
odds_ratio_stem = odds_stem / (1 - odds_stem)


odds_non_stem = (number_non_stem_with_link ) / nrow(okc_sampled)
odds_ratio_non_stem = odds_non_stem / (1 - odds_non_stem)

#Confidence interval for odds ratio

odds_ratio_matrix = matrix(c(number_stem_with_link, nrow(stem_data) - number_stem_with_link, number_non
odds_ratio_matrix
```

```
##      [,1] [,2]
## [1,]  409  253
## [2,] 4591 4747
```

```r
oddsratio.wald(odds_ratio_matrix)
```

```
## $data
##           Outcome
## Predictor  Disease1 Disease2 Total
##   Exposed1      409      253   662
##   Exposed2     4591     4747  9338
##   Total        5000     5000 10000
##
## $measure
##           odds ratio with 95% C.I.
## Predictor  estimate    lower    upper
##   Exposed1 1.000000       NA       NA
##   Exposed2 1.671532 1.421637 1.965354
##
## $p.value
##           two-sided
## Predictor    midp.exact fisher.exact   chi.square
##   Exposed1           NA           NA           NA
##   Exposed2 3.051097e-10 3.835504e-10 3.511003e-10
##
## $correction
## [1] FALSE
##
## attr(,"method")
## [1] "Unconditional MLE & normal approximation (Wald) CI"
```

```
library(caret)
library(tidymodels)
library(keras)
library(doParallel)
library(pROC)

load("./Datasets/okc.RData")
load("./Datasets/okc_other.RData")
load("./Datasets/okc_binary.RData")
load("./Datasets/okc_features.RData")

# joining all pre-computed features data sets: joins basic dataset, binary dataset and basic precompute
okc_train = okc_train %>% full_join(okc_train_binary) %>%full_join(basic_features_train) %>% arrange(pr

# Selecting pre-computed textual features names.
text_features = c("n_urls", "n_hashtags", "n_mentions", "n_commas", "n_digits",
    "n_exclaims", "n_extraspaces", "n_lowers", "n_lowersp", "n_periods",
    "n_words", "n_puncts", "n_charsperword")

#Specifying sentence features names.
sentiment_features = c("sent_afinn", "sent_bing", "n_polite", "n_first_person", "n_first_personp",
    "n_second_person", "n_second_personp", "n_third_person", "n_prepositions")

# Getting data set base features names.
base_features = names(okc_test)
base_features = base_features[!(base_features %in% c('Class','profile','essay_length','where_state'))]

#Getting pre-computed keyword features names.
keyword_features = names(okc_train_binary)
keyword_features = keyword_features[keyword_features != 'profile']

#Function used to get useful statistics from data and specified model
statistiche = function(data, lev= levels(data$obs), model = NULL){
  c(
    twoClassSummary(data = data, lev = levels(data$obs), model),
    prSummary(data = data, lev = levels(data$obs), model),
    mnLogLoss(data = data, lev = levels(data$obs), model),
    defaultSummary(data = data, lev = levels(data$obs), model)
  )
}

# used to get computational data of the models
okc_train_control = trainControl(
  method = "cv",
  classProbs = TRUE,
  summaryFunction = statistiche,
  returnData = FALSE,
  trim = TRUE,
  sampling = "down"
)

# keyword normalization
keyword_normalization =
```

```
  #transforms dataset to recipe object with basic and keyword features
  recipe(Class ~.,data=okc_train[, c("Class", base_features, keyword_features)]) %>%
  step_YeoJohnson(all_numeric()) %>%
  step_other(where_town) %>%
  # Creates dummy variables when needed
  step_dummy(all_nominal(), -all_outcomes()) %>%
  # Remove zero variance predictors
  step_zv(all_predictors()) %>%
  #Centers all predictors
  step_center(all_predictors()) %>%
  # Scale all predictors to have a std dev equal to 1
  step_scale(all_predictors())

okc_control_rand = okc_train_control
okc_control_rand$search = "random"

set.seed(49)

#multi_layer_perceptron_keyword = train(
#  keyword_normalization,
#  data = okc_train,
#  method = "mlpKerasDropout",
#  metric= "ROC",
#  tuneLength = 20,
#  trControl=okc_control_rand,
#  verbose = 0,
#  epochs = 500
#)



#Model with basic profile info

basic_model =
  recipe(Class ~ ., data=okc_train[, c('Class', base_features)]) %>%
  step_YeoJohnson(all_numeric()) %>%
  step_other(where_town) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

set.seed(49)

glm_basic <- train(
  basic_model,
  data = okc_train,
  method = "glm",
  metric = "ROC",
  trControl = okc_train_control
)
basic_pred <- ncol(glm_basic$recipe$template) - 1

#Model with basic text features
text_rec <-
```

```
  recipe(Class ~ .,
         data = okc_train[, c("Class", base_features, text_features)]) %>%
  step_YeoJohnson(all_numeric()) %>%
  step_other(where_town) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

set.seed(49)
glm_text <- train(
  text_rec,
  data = okc_train,
  method = "glm",
  metric = "ROC",
  trControl = okc_train_control
)

text_pred <- ncol(glm_text$recipe$template) - 1

#Model with base info, text features and keywords
keyword_rec <-
  recipe(Class ~ .,
         data = okc_train[, c("Class", base_features, text_features, keyword_features)]) %>%
  step_YeoJohnson(all_numeric()) %>%
  step_other(where_town) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

okc_ctrl_keep <- okc_train_control
okc_ctrl_keep$savePredictions <- "final"

set.seed(49)
glm_keyword <- train(
  keyword_rec,
  data = okc_train,
  method = "glm",
  metric = "ROC",
  trControl = okc_ctrl_keep
)

keyword_pred <- ncol(glm_keyword$recipe$template) - 1

#Model with base info, text features, keywords and sentiment analysis

sent_rec <-
  recipe(Class ~ .,
         data = okc_train[, c("Class", base_features, keyword_features, sentiment_features)]) %>%
  step_YeoJohnson(all_numeric()) %>%
  step_other(where_town) %>%
  step_dummy(all_nominal(), -all_outcomes()) %>%
  step_zv(all_predictors())

set.seed(49)
glm_sent <- train(
```

```
  sent_rec,
  data = okc_train,
  method = "glm",
  metric = "ROC",
  trControl = okc_train_control
)

sent_pred <- ncol(glm_sent$recipe$template) - 1

#Data collection and visualization
features_groups <-
  c("Basic Profile Info", "+ Simple Text Features", "+ Keywords", "+ Keywords\n+ Sentiment")

glm_resamples <-
  glm_basic$resample %>%
  mutate(Features = "Basic Profile Info") %>%
  bind_rows(
    glm_text$resample %>%
      mutate(Features = "+ Simple Text Features"),
    glm_keyword$resample %>%
      mutate(Features = "+ Keywords"),
    glm_sent$resample %>%
      mutate(Features = "+ Keywords\n+ Sentiment")
  ) %>%
  mutate(Features = factor(Features, levels = features_groups))

glm_resamples_mean <-
  glm_resamples %>%
  group_by(Features) %>%
  summarize(ROC = mean(ROC))
glm_resamples_mean$Resample = 'Average'

temp_df = glm_resamples[,-c(2,3,4,5,6,7,8,9,10)]

full_df = rbind(glm_resamples_mean,temp_df )
library(ggplot2)


ggplot(data = subset(full_df, Resample != 'Average'),aes(x=Features, y=ROC, colour=Resample, group=Resa
```