Data

The data for this article was prepared synthetically and the code to prepare it can be found in the code “01_Synthetic_Data_Preparation.R” in the repository. It comprises 5,000 observations and three variables

- *city* – A categorical variable whose values may be one of 100 unique American city names obtained from the *world.cities* dataset available in the *maps* package. Some observations may have missing values for this variable.
- *hml* – A categorical variable whose values are ‘High’, ‘Medium’ or ‘Low’. The intent is to demonstrate an ordinal feature. Some observations may have missing values for this variable.
- *target* – A numeric variable which may take one of two values 0 or 1. This can be considered as the dependent variable for a binary classification problem. There are no missing values for this variable.

There is also a relationship between the *hml* and *target* variables, with the value *High* having the lowest target rate (percentage of records where the target is 1) and the value *Low* having the highest target rate.

The data is referred to by the variable *df* in the example code below.

Encoders – A Short Summary

The encoders described in this article can be divided into two key groups

- Classic and Contrast encoders – These do not use the information from the dependent variable in the encoding. If there are *k* unique values in a categorical variable, they create up to *k* distinct columns to store the encoding depending on the technique.
- Bayesian encoders – These use information from the dependent variable in the encoding. One must ensure that the encoding is only done after splitting the data into training and test sets and using the target from the training set. These encoders always create one column to store the encoded value. It is often recommended to add some ‘noise’ to the resulting encoded value.

Some encoders are applicable mostly for ordinal variables while others can be used for both ordinal and nominal variables. In this article, we will not get into the question of deciding which encoding is the most appropriate. There are already excellent resources available in the internet on this topic and a quick web search should lead you to interesting articles or discussions for that particular encoding technique.

The classic and contrast encoders described are

- Ordinal
- One-Hot
- Binary
- Hashing
- Helmert
- Backward difference
- Polynomial

The Bayesian encoders described in the article are

- Target
- Leave one out
- Weight of evidence
- James Stein
- M Estimator

Classic and Contrast Encoders

Ordinal

This is the simplest form of encoding where each value is converted to an integer. The maximum value is the equal to the number of unique values of the variable. By default, the values are considered in the order in which they appear in the column but can be changed using the *order* argument to the function. The *exclude* argument is needed in the *factor* function to ensure that missing values are also assigned an encoded value.

```
encode_ordinal <- function(x, order = unique(x)) {
  x <- as.numeric(factor(x, levels = order, exclude = NULL))
  x
}
```

A few examples to demonstrate the function in action. The order in which the values occur in the data is *Medium*, *Low*, *High* and *NA*. So the original call to the function assigns the values 1 to 4 in this order. This can be changed by the user passing an explicit order. In general, encoding a nominal variable like *city* is not recommended using this technique and the code has been used for illustration only.

```
table(df[["hml"]], encode_ordinal(df[["hml"]]), useNA = "ifany")
```

```
##
##             1    2    3    4
##   High      0    0 1618    0
##   Low       0 1545    0    0
##   Medium 1587    0    0    0
##          0    0    0  250
```

```
table(df[["hml"]],
      encode_ordinal(df[["hml"]], order = c(NA, "Low", "Medium", "High")),
      useNA = "ifany")
```

```
##
##             1    2    3    4
##   High      0    0    0 1618
##   Low       0 1545    0    0
##   Medium    0    0 1587    0
##        250    0    0    0
```

```
table(df[["hml"]],
      encode_ordinal(df[["hml"]], order = c("Low", "Medium", "High", NA)),
      useNA = "ifany")
```

```
##
##             1    2    3    4
##   High      0    0 1618    0
##   Low    1545    0    0    0
##   Medium    0 1587    0    0
##          0    0    0  250
```

```
new_df <- df
new_df[["hml_encoded"]] <- encode_ordinal(df[["hml"]])
new_df[["city_encoded"]] <- encode_ordinal(df[["city"]])
head(new_df)
```

```
##             city    hml target hml_encoded city_encoded
## 1:      Hartford Medium      0           1            1
## 2: Athens-Clarke    Low      0           2            2
## 3:         Pasco    Low      1           2            3
## 4:        Nashua   High      0           3            4
## 5:    Bellflower   High      0           3            5
## 6:       Kendall    Low      1           2            6
```

One-Hot

This is probably the most common form of encoding and is often referred to as creating dummy or indicator variables.

It creates a new column for each unique value of the categorical variable. Each of these columns are binary with values 1 or 0 depending on whether the value of the variable is equal to the unique value being encoded by this column.

There are multiple ways to do one-hot encoding in R. One way would be to use the *model.matrix* function from the *stats* package.

```
# Using model.matrix
new_df <- df
new_df$city <- factor(new_df$city, exclude = NULL)
new_df$hml <- factor(new_df$hml, exclude = NULL)
new_df <- model.matrix(~.-1, data = new_df[, c("city", "hml")],
                       contrasts.arg = list(
                         city = contrasts(new_df$city, contrasts = FALSE),
                         hml = contrasts(new_df$hml, contrasts = FALSE)
                       ))
head(new_df[, 1:3])
```

```
##   cityAlameda cityAmes cityApple Valley
## 1           0        0               0
## 2           0        0               0
## 3           0        0               0
## 4           0        0               0
## 5           0        0               0
## 6           0        0               0
```

I would personally recommend using the *designTreatmentsZ* and *prepare* functions from the *vtreat* package. While a comprehensive discussion of *vtreat* is beyond the scope of this article, it has a lot of nice features and takes care of multiple potential issues for you.

```
library(vtreat)
tz <- vtreat::designTreatmentsZ(df, c("city", "hml"))
```

```
## [1] "vtreat 1.5.1 inspecting inputs Mon Feb 03 00:16:33 2020"
## [1] "designing treatments Mon Feb 03 00:16:33 2020"
## [1] " have initial level statistics Mon Feb 03 00:16:33 2020"
## [1] " scoring treatments Mon Feb 03 00:16:33 2020"
## [1] "have treatment plan Mon Feb 03 00:16:34 2020"
```

```
new_df <- vtreat::prepare(tz, df, extracols = "target")
head(new_df[, 1:5])
```

```
##   city_catP hml_catP city_lev_NA city_lev_x_Alameda city_lev_x_Ames
## 1    0.0090   0.3174           0                  0               0
## 2    0.0094   0.3090           0                  0               0
## 3    0.0106   0.3090           0                  0               0
## 4    0.0116   0.3236           0                  0               0
## 5    0.0106   0.3236           0                  0               0
## 6    0.0094   0.3090           0                  0               0
```

```
detach("package:vtreat", unload = TRUE)
```

The third option is to use the *dummyVars* function from the *caret* package. This uses the R formula interface and one needs to use the *predict* function to actually create the variables.

```
library(caret)
```

```
## Loading required package: lattice
```

```
## Loading required package: ggplot2
```

```
new_df <- df
new_df$city <- factor(new_df$city, exclude = NULL)
new_df$hml <- factor(new_df$hml, exclude = NULL)
new_df$city <- addNA(new_df$city)
new_df$hml <- addNA(new_df$hml)
dv <- caret::dummyVars(" ~ city + hml", data = new_df)
new_df <- data.frame(predict(dv, newdata = df))
head(new_df[, 1:3])
```

```
##   city.Alameda city.Ames city.Apple.Valley
## 1            0         0                 0
## 2            0         0                 0
## 3            0         0                 0
## 4            0         0                 0
## 5            0         0                 0
## 6            0         0                 0
```

```
detach("package:caret", unload = TRUE)
```

Binary

Instead of each unique value being assigned to an integer, the integers are converted to a binary number and a representation of this binary number is stored as multiple columns, where each column can take the values 0 or 1. In that sense, it can be thought of as a combination of ordinal and one-hot encoding. So if you have 27 distinct values of a categorical variable, then 5 columns are sufficient to encode this variable – as 5-digit binary numbers can store any value from 0 to 31. An implementation is provided below using the *binaryLogic* package. We first use ordinal encoding. The *as.binary* function from this package is used to get the binary representation of the integer assigned to that category. As not all numbers will have the same length of representation, the numbers are “padded by 0s”. From the example, we see that 7 indicator variables are sufficient to represent all the 100 cities as well as the missing value in the data.

```
library(binaryLogic)
encode_binary <- function(x, order = unique(x), name = "v_") {
  x <- as.numeric(factor(x, levels = order, exclude = NULL))
  x2 <- as.binary(x)
  maxlen <- max(sapply(x2, length))
  x2 <- lapply(x2, function(y) {
    l <- length(y)
    if (l < maxlen) {
      y <- c(rep(0, (maxlen - l)), y)
    }
    y
  })
  d <- as.data.frame(t(as.data.frame(x2)))
  rownames(d) <- NULL
  colnames(d) <- paste0(name, 1:maxlen)
  d
}

new_df <- cbind(df, encode_binary(df[["city"]], name = "city_"))
head(new_df)
```

```
##             city    hml target city_1 city_2 city_3 city_4 city_5 city_6 city_7
## 1:      Hartford Medium      0      0      0      0      0      0      0      1
## 2: Athens-Clarke    Low      0      0      0      0      0      0      1      0
## 3:         Pasco    Low      1      0      0      0      0      0      1      1
## 4:        Nashua   High      0      0      0      0      0      1      0      0
## 5:    Bellflower   High      0      0      0      0      0      1      0      1
## 6:       Kendall    Low      1      0      0      0      0      1      1      0
```

```
detach("package:binaryLogic", unload = TRUE)
```

Hashing

This uses the hashing trick described in this article.

An implementation of this technique is provided by the *FeatureHashing* package. The *hash.size* should be chosen so that the collision rate is not too high. A simple example of creating the features and using it in *xgboost* is also shown in the example below.

```
library(FeatureHashing)
library(xgboost)
m.mat <- hashed.model.matrix(c("city", "hml"), df, hash.size = 2 ^ 10,
                             create.mapping = TRUE)

# Extract the mapping
mapping <- hash.mapping(m.mat)

# Check collision rate
mean(duplicated(mapping))
```

```
## [1] 0.05825243
```

```
# Names
names(mapping)
```

```
##   [1] "cityIndio"                "cityCherry Hill"         "hmlHigh"
##   [4] "cityHattiesburg"          "cityRockford"            "cityLynwood"
##   [7] "cityEdinburg"             "cityAustin"              "cityPleasanton"
##  [10] "cityWest Orange"          "cityWilkes-Barre"        "cityHaverhill"
##  [13] "cityPeoria"               "cityBellflower"          "hmlMedium"
##  [16] "cityDeKalb"               "hmlLow"                  "cityCitrus Heights"
##  [19] "cityRacine"               "citySomerville"          "cityFlorence-Graham"
##  [22] "cityYorba Linda"          "cityWilson"              "cityVacaville"
##  [25] "cityHamilton"             "cityChandler"            "cityPocatello"
##  [28] "cityHartford"             "cityDetroit"             "cityFranconia"
##  [31] "citySaint Louis Park"     "cityElizabeth"           "cityPearland"
##  [34] "cityOverland Park"        "cityFindlay"             "cityApple Valley"
##  [37] "cityNewport News"         "cityMesa"                "cityHemet"
##  [40] "cityPassaic"              "cityCharleston"          "cityBlacksburg"
##  [43] "citySioux City"           "cityCary"                "cityChico"
##  [46] "cityCrystal Lake"         "cityBolingbrook"         "cityWalnut Creek"
##  [49] "cityWestminster"          "cityWaterbury"           "cityFond du Lac"
##  [52] "citySalt Lake City"       "cityBridgewater"         "cityCarson"
##  [55] "cityWheaton-Glenmont"     "cityNorth Bethesda"      "cityWatsonville"
##  [58] "cityGrand Island"         "cityNashua"              "cityDothan"
##  [61] "cityKalamazoo"            "cityEllicott City"       "cityEau Claire"
##  [64] "cityGilbert"              "cityPasco"               "cityHoboken"
##  [67] "cityMontgomery Village" "cityHilo"                  "cityJupiter"
##  [70] "cityTopeka"               "cityPittsburgh"          "cityGlendora"
##  [73] "cityBoynton Beach"        "cityFort Myers"          "cityWestland"
##  [76] "cityYonkers"              "cityFort Worth"          "cityMinnetonka"
##  [79] "citySarasota"             "cityConcord"             "cityBattle Creek"
##  [82] "cityWest Sacramento"      "cityShelby"              "cityAlameda"
##  [85] "cityBroken Arrow"         "cityJoliet"              "cityGarden Grove"
##  [88] "citySan Clemente"         "cityPharr"               "cityAmes"
##  [91] "cityTurlock"              "cityFort Smith"          "cityWoodland"
##  [94] "citySan Buenaventura"     "cityBeaverton"           "cityFort Wayne"
##  [97] "cityAthens-Clarke"        "cityMansfield"           "cityMinneapolis"
## [100] "cityToledo"              "cityKendall"             "cityBloomfield"
## [103] "cityBryan"
```

```
# Hashed values
hashed.value(names(mapping))
```

```
##   [1] -1783827961 -1554688119 -1885087034   903710395  -433534873 -1698157690
##   [7]  1000408268 -1738513785   327611540   744213640   101526589   -97403553
##  [13]   305346091 -1622681438 -1895904773 -1797244496  -775669476 -1581269032
##  [19]  -583515489   823993455  -475980681  1164925682 -1986211414 -1614513926
##  [25]   664810711  -283418510 -1627492277   594557760  -873606777 -1364185228
##  [31] -1160352055 -1874159752 -1049081229   523278412 -1020228123  1555812291
##  [37] -1468685840 -1311498863   407357228  2002198630  1444597631   176643076
##  [43]  1486456091  -489505536   -40219489    86822926  1912518627  1934415346
##  [49]   229200030  1718684157  -212552741 -1918227473   356556145  -434725725
##  [55]   311621850  1373663104  1322620681   285510556  1916763754   273751018
##  [61]  1583117190   949246687 -1883264844 -1177970518  1718097732  -951650603
##  [67] -1766063377 -1433688539  1350976259  -563052972  1423031903 -1800689849
##  [73]   232908642 -1215777705  1600270663   266027690 -1060761545 -1107504569
##  [79]  -588399577  1817885574 -1773860050  -756120522   743992857  1583240399
##  [85]  -741470805  1905324187   320817142 -2121018767 -1079400486  1445605708
##  [91]  -676887260   221591167 -1991619194    20030044  -912243463   172128751
##  [97] -1909238049 -1076346490 -1617920854 -1900950692  1677793300  -822743039
## [103]  -201665694
```

```
# Testing with xgboost
bst <- xgboost(data = m.mat, label = df$target, nround = 10, params = list(
  booster = "gbtree",
  objective = "binary:logistic",
  eval_metric = "auc"
))
```

```
## [1]  train-auc:0.785905
## [2]  train-auc:0.792083
## [3]  train-auc:0.796824
## [4]  train-auc:0.798893
## [5]  train-auc:0.799901
## [6]  train-auc:0.801744
## [7]  train-auc:0.802468
## [8]  train-auc:0.803876
## [9]  train-auc:0.805856
## [10] train-auc:0.806401
```

```
detach("package:xgboost", unload = TRUE)
detach("package:FeatureHashing", unload = TRUE)
```

Helmert

Helmert encoding is a form of contrast encoding where each value of a categorical variable is compared to the mean of the subsequent levels. A nice explanation can be found in this StackOverlow answer by user *StatsStudent* and the comments by user *whuber*. The implementation below is inspired by one of the code samples from *whuber*, while keeping it consistent with the values output by the *category_encoders* module for use with *scikit-learn* in Python. The output of the *encode_helmert* function below is a data frame which can be merged back to the original data to get the encoding for all values in the data.

```
helmert <- function(n) {
  m <- t((diag(seq(n-1, 0)) - upper.tri(matrix(1, n, n)))[-n,])
  t(apply(m, 1, rev))
}
```

```
encode_helmert <- function(df, var) {
  x <- df[[var]]
  x <- unique(x)
  n <- length(x)
  d <- as.data.frame(helmert(n))
  d[[var]] <- rev(x)
  names(d) <- c(paste0(var, 1:(n-1)), var)
  d
}

d <- encode_helmert(df, "hml")
d
```

```
##   hml1 hml2 hml3    hml
## 1    0    0    3       
## 2    0    2   -1   High
## 3    1   -1   -1    Low
## 4   -1   -1   -1 Medium
```

Backward Difference

This is similar to Helmert encoding, except that in this system, the mean for one level of the categorical variable is compared to the mean for the prior adjacent level.

```
backward_difference <- function(n) {
  m <- matrix(0:(n-1), nrow = n, ncol = n)
  m <- m + upper.tri(matrix(1, n, n))
  m2 <- matrix(-(n-1), n, n)
  m2[upper.tri(m2)] <- 0
  m <- (m + m2) / n
  m <- (t(m))[, -n]
  m
}

encode_backward_difference <- function(df, var) {
  x <- df[[var]]
  x <- unique(x)
  n <- length(x)
  d <- as.data.frame(backward_difference(n))
  d[[var]] <- x
  names(d) <- c(paste0(var, 1:(n-1)), var)
  d
}

d <- encode_backward_difference(df, "hml")
d
```

```
##    hml1 hml2  hml3    hml
## 1 -0.75 -0.5 -0.25 Medium
## 2  0.25 -0.5 -0.25    Low
## 3  0.25  0.5 -0.25   High
## 4  0.25  0.5  0.75       
```

Polynomial

It is again similar to the two contrast encoders above; it looks for the linear, quadratic and cubic trends in a categorical variable. The implementation of this encoding is simple using the *contr.poly* function from the *stats* package.

```
encode_polynomial <- function(df, var) {
  x <- df[[var]]
  x <- unique(x)
  n <- length(x)
  d <- as.data.frame(contr.poly(n))
  d[[var]] <- x
  names(d) <- c(paste0(var, 1:(n-1)), var)
  d
}

d <- encode_polynomial(df, "hml")
d
```

```
##         hml1 hml2       hml3    hml
## 1 -0.6708204  0.5 -0.2236068 Medium
## 2 -0.2236068 -0.5  0.6708204    Low
## 3  0.2236068 -0.5 -0.6708204   High
## 4  0.6708204  0.5  0.2236068
```

Bayesian Encoders

Target

Target encoding is also very simple, where the encoded value of each value of a categorical variable is simply the mean of the target variable. The mean of the target is obtained by using the *aggregate* R function. Some noise can be added to the encoded value by specifying the *sigma* argument. If specified, it draws a random sample from the normal distribution with mean 1 and standard deviation equal to *sigma*, and multiplies the mean of the target with this random value. Note that the addition of this noise is usually done only on the training set and not the test set.

```
encode_target <- function(x, y, sigma = NULL) {
  d <- aggregate(y, list(factor(x, exclude = NULL)), mean, na.rm = TRUE)
  m <- d[is.na(as.character(d[, 1])), 2]
  l <- d[, 2]
  names(l) <- d[, 1]
  l <- l[x]
  l[is.na(l)] <- m
  if (!is.null(sigma)) {
    l <- l * rnorm(length(l), mean = 1, sd = sigma)
  }
  l
}

table(encode_target(df[["hml"]], df[["target"]]), df[["hml"]], useNA = "ifany")
```

```
##
##                      High  Low Medium
##   0.0247218788627936 1618    0      0    0
##   0.0945179584120983    0    0   1587    0
##   0.323624595469256     0 1545      0    0
##   0.5                   0    0      0  250
```

```
new_df <- df
new_df[["hml_encoded"]] <- encode_target(df[["hml"]], df[["target"]])
new_df[["hml_encoded2"]] <- encode_target(df[["hml"]], df[["target"]],
                                          sigma = 0.05)
new_df[["city_encoded"]] <- encode_target(df[["city"]], df[["target"]])
head(new_df)
```

```
##             city    hml target hml_encoded hml_encoded2 city_encoded
## 1:       Hartford Medium      0  0.09451796   0.09918551    0.3111111
## 2: Athens-Clarke     Low      0  0.32362460   0.33714512    0.2553191
## 3:          Pasco     Low      1  0.32362460   0.31293654    0.1132075
## 4:         Nashua    High      0  0.02472188   0.02665522    0.0862069
## 5:     Bellflower    High      0  0.02472188   0.02271573    0.1320755
## 6:        Kendall     Low      1  0.32362460   0.33760350    0.1914894
```

Leave One Out

This is very similar to target encoding, except that for each row we exclude that row in calculating the mean of the target. Some noise can be added to the encoded value by specifying the *sigma* argument. In the implementation below, this is done by iterating over each row and removing it prior to calculating the mean of the target. For binary targets, the implementation can be significantly optimized by noting that we can pre-calculate the totals by each unique value and the target. Removing that row is then only subtracting from the total for that value and target combination (the implementation is left as an exercise to the reader).

```
encode_leave_one_out <- function(x, y, sigma = NULL) {
  n <- length(x)
  x[is.na(x)] <- "__MISSING"
  x2 <- vapply(1:n, function(i) {
    xval <- x[i]
    yloo <- y[-i]
    xloo <- x[-i]
    yloo <- yloo[xloo == xval]
    mean(yloo, na.rm = TRUE)
  }, numeric(1))
  if (!is.null(sigma)) {
    x2 <- x2 * rnorm(n, mean = 1, sd = sigma)
  }
  x2
}

new_df <- df
new_df[["hml_encoded"]] <- encode_leave_one_out(df[["hml"]], df[["target"]])
set.seed(1123)
new_df[["city_encoded"]] <- encode_leave_one_out(df[["city"]], df[["target"]],
                                                  sigma = 0.05)
head(new_df)
```

```
##              city    hml target hml_encoded city_encoded
## 1:       Hartford Medium      0  0.09457755   0.32939353
## 2: Athens-Clarke     Low      0  0.32383420   0.27531012
## 3:          Pasco     Low      1  0.32318653   0.10067333
## 4:         Nashua    High      0  0.02473717   0.08372383
## 5:     Bellflower    High      0  0.02473717   0.14392368
## 6:        Kendall     Low      1  0.32318653   0.17560372
```

Weight of Evidence

Weight of evidence encoding is very popular in the field of credit scoring (credit risk modelling). In this, the encoded value is the logarithm of the odds of the target. The odds is defined to be the percentage of records with target value 0 divided by the percentage of records with target value 1. The implementation below is valid only for binary targets. The formula to calculate weight of evidence can also be modified for continous target (see link), but that implementation is left as an exercise to the reader. In the field of credit scoring, it is also common practice to group related values together and then calculate the weight of evidence. For example, if there is a field called *marital_status*, one may decide to combine the values of *Divorced* and *Separated* before calculating weight of evidence if the empirical data suggests that they have very similar target rates.

```
encode_woe <- function(x, y, sigma = NULL) {
  d <- aggregate(y, list(factor(x, exclude = NULL)), mean, na.rm = TRUE)
  d[["woe"]] <- log(((1 / d[, 2]) - 1) *
                      (sum(y) / sum(1-y)))
  m <- d[is.na(as.character(d[, 1])), 3]
  l <- d[, 3]
  names(l) <- d[, 1]
  l <- l[x]
  l[is.na(l)] <- m
  if (!is.null(sigma)) {
    l <- l * rnorm(length(l), mean = 1, sd = sigma)
  }
  l
}

table(encode_woe(df[["hml"]], df[["target"]]), df[["hml"]], useNA = "ifany")
```

```
##
##                       High  Low Medium
##   -1.63607386968271      0    0      0  250
##   -0.898909803705993     0 1545      0    0
##   0.623603722300056      0    0   1587    0
##   2.03896017760917    1618    0      0    0
```

```
new_df <- df
new_df[["hml_encoded"]] <- encode_woe(df[["hml"]], df[["target"]])
new_df[["hml_encoded2"]] <- encode_woe(df[["hml"]], df[["target"]],
                                       sigma = 0.05)
new_df[["city_encoded"]] <- encode_woe(df[["city"]], df[["target"]])
head(new_df)
```

```
##              city    hml target hml_encoded hml_encoded2 city_encoded
## 1:       Hartford Medium      0   0.6236037    0.6259107   -0.8411440
## 2: Athens-Clarke    Low      0  -0.8989098   -0.8513090   -0.5656325
## 3:          Pasco    Low      1  -0.8989098   -0.8585435    0.4223143
## 4:         Nashua   High      0   2.0389602    2.0354463    0.7247801
## 5:     Bellflower   High      0   2.0389602    2.1024377    0.2466574
## 6:        Kendall    Low      1  -0.8989098   -0.8857241   -0.1957123
```

James Stein

A good explanation of this encoding can be found in this article. This method is similar to target encoding but the encoded value is shrunk towards the overall population mean of the target. So the encoded value is a weighted sum of the target mean for that category and the population mean.

```
encode_james_stein <- function(x, y, sigma = NULL) {
  n_all <- length(y)
  p_all <- mean(y)
  var_all <- (p_all * (1 - p_all)) / n_all

  d <- aggregate(y, list(factor(x, exclude = NULL)), mean, na.rm = TRUE)
  d2 <- aggregate(y, list(factor(x, exclude = NULL)), length)
  g <- names(d)[1]
  d <- merge(d, d2, by = g, all = TRUE)
  d[, 4] <- (d[, 2] * (1 - d[, 2])) / d[, 3]
  d[, 5] <- d[, 4] / (d[, 4] + var_all)
  d[, 6] <- (1 - d[, 5]) * d[, 2] + d[, 5] * p_all

  m <- d[is.na(as.character(d[, 1])), 6]
  l <- d[, 6]
  names(l) <- d[, 1]
  l <- l[x]
  l[is.na(l)] <- m
  if (!is.null(sigma)) {
    l <- l * rnorm(length(l), mean = 1, sd = sigma)
  }
  l
}

table(encode_james_stein(df[["hml"]], df[["target"]]),
      df[["hml"]], useNA = "ifany")

##
##                       High  Low Medium
##   0.0735644532521975 1618    0      0    0
##   0.13999163072412      0    0   1587    0
##   0.171951205029329     0    0      0  250
##   0.18893950763687      0 1545      0    0

new_df <- df
new_df[["hml_encoded"]] <- encode_james_stein(df[["hml"]], df[["target"]])
new_df[["hml_encoded2"]] <- encode_james_stein(df[["hml"]], df[["target"]],
                                               sigma = 0.05)
new_df[["city_encoded"]] <- encode_james_stein(df[["city"]], df[["target"]])
```

M Estimator

It is similar to the James Stein encoder where the target mean for each unique value is adjusted using the population mean using one additional parameter *m*. The default value of *m* is 1.

```
encode_m_estimator <- function(x, y, m = 1, sigma = NULL) {
  p_all <- mean(y)

  d <- aggregate(y, list(factor(x, exclude = NULL)), sum, na.rm = TRUE)
  d2 <- aggregate(y, list(factor(x, exclude = NULL)), length)
  g <- names(d)[1]
  d <- merge(d, d2, by = g, all = TRUE)
  d[, 4] <- (d[, 2] + p_all * m) / (d[, 3] + m)

  m <- d[is.na(as.character(d[, 1])), 4]
  l <- d[, 4]
  names(l) <- d[, 1]
  l <- l[x]
  l[is.na(l)] <- m
  if (!is.null(sigma)) {
    l <- l * rnorm(length(l), mean = 1, sd = sigma)
  }
  l
}

table(encode_m_estimator(df[["hml"]], df[["target"]]),
      df[["hml"]], useNA = "ifany")

##
##                       High  Low Medium
##   0.0248072884496603 1618    0      0    0
##   0.0945610831234257    0    0   1587    0
##   0.323520698576973     0 1545      0    0
##   0.498657370517928     0    0      0  250
```

```
new_df <- df
new_df[["hml_encoded"]] <- encode_m_estimator(df[["hml"]], df[["target"]])
new_df[["hml_encoded2"]] <- encode_m_estimator(df[["hml"]], df[["target"]],
                                              sigma = 0.05)
new_df[["city_encoded"]] <- encode_m_estimator(df[["city"]], df[["target"]])
```