# TensorFlow feature columns: Transforming your data recipes-style

TENSORFLOW/KERAS TABULAR DATA

TensorFlow feature columns provide useful functionality for preprocessing categorical data and chaining transformations, like bucketization or feature crossing. From R, we use them in popular "recipes" style, creating and subsequently refining a feature specification. In this post, we show how using feature specs frees cognitive resources and lets you focus on what you really want to accomplish. What's more, because of its elegance, feature-spec code reads nice and is fun to write as well.

AUTHORS
Daniel Falbel
Sigrid Keydana

AFFILIATIONS
RStudio
RStudio

PUBLISHED
July 8, 2019

CITATION
Falbel & Keydana, 2019

It's 2019; no one doubts the effectiveness of deep learning in computer vision. Or natural language processing. With "normal," Excel-style, a.k.a. *tabular* data however, the situation is different.

Basically there are two cases: One, you have numeric data only. Then, creating the network is straightforward, and all will be about optimization and hyperparameter search. Two, you have a mix of numeric and categorical data, where categorical could be anything from ordered-numeric to symbolic (e.g., text). In this latter case, with categorical data entering the picture, there is an extremely nice idea you can make use of: *embed* what are equidistant symbols into a high-dimensional, numeric representation. In that new representation, we can define a distance metric that allows us to make statements like "cycling is closer to running than to baseball," or "😃 is closer to 😂 than to 😠." When not dealing with language data, this technique is referred to as *entity embeddings*. [1]

Nice as this sounds, why don't we see entity embeddings used all the time? Well, creating a Keras network that processes a mix of numeric and categorical data used to require a bit of an effort. With TensorFlow's new *feature columns*, usable from R through a combination of `tfdatasets` and `keras`, there is a much easier way to achieve this. What's more, `tfdatasets` follows the popular recipes idiom to initialize, refine, and apply a feature specification `%>%`-style. And finally, there are ready-made steps for bucketizing a numeric column, or hashing it, or creating *crossed columns* to capture interactions.

This post introduces feature specs starting from a scenario where they don't exist: basically, the status quo until very recently. Imagine you have a dataset like that from the Porto Seguro

car insurance competition where some of the columns are numeric, and some are

categorical. You want to train a fully connected network on it, with all categorical columns fed into embedding layers. How can you do that? We then contrast this with the feature spec way, which makes things *a lot* easier – especially when there's a lot of categorical columns. In a second applied example, we demonstrate the use of *crossed columns* on the *rugged* dataset from Richard McElreath's *rethinking* package. Here, we also direct attention to a few technical details that are worth knowing about.

# Mixing numeric data and embeddings, the pre-feature-spec way

Our first example dataset is taken from Kaggle. Two years ago, Brazilian car insurance company Porto Seguro asked participants to predict how likely it is a car owner will file a claim based on a mix of characteristics collected during the previous year. The dataset is comparatively large – there are ~ 600,000 rows in the training set, with 57 predictors. Among others, features are named so as to indicate the type of the data – binary, categorical, or continuous/ordinal. While it's common in competitions to try to reverse-engineer column meanings, here we just make use of the type of the data, and see how far that gets us.

Concretely, this means we want to

- use binary features just the way they are, as zeroes and ones,
- scale the remaining numeric features to mean 0 and variance 1, and
- embed the categorical variables (each one by itself).

We'll then define a dense network to predict `target`, the binary outcome. So first, let's see how we could get our data into shape, as well as build up the network, in a "manual," pre-feature-columns way.

When loading libraries, we already use the versions we'll need very soon: Tensorflow 2 (>= beta 1), and the development (= Github) versions of `tfdatasets` and `keras`:

```
tensorflow::install_tensorflow(version = "2.0.0-beta1")

remotes::install_github("rstudio/tfdatasets")
remotes::install_github("rstudio/keras")

library(keras)
library(tfdatasets)
library(readr)
library(dplyr)
```

In this first version of preparing the data, we make our lives easier by assigning different R types, based on what the features represent (categorical, binary, or numeric qualities): [2]

```
# downloaded from https://www.kaggle.com/c/porto-seguro-safe-driver-
    prediction/data
path <- "train.csv"

porto <- read_csv(path) %>%
  select(-id) %>%
  # to obtain number of unique levels, later

  mutate_at(vars(ends_with("cat")),
```

```
  factor  )        %>%
    # to easily keep them apart from the non-binary numeric data
    mutate_at(       vars    (       ends_with(       "bin"    )       )       ,
  as.integer)

  porto    %>%      glimpse (       )
```

```
Observations: 595,212
Variables: 58
$ target         <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,…
$ ps_ind_01      <dbl> 2, 1, 5, 0, 0, 5, 2, 5, 5, 1, 5, 2, 2, 1, 5, 5,…
$ ps_ind_02_cat  <fct> 2, 1, 4, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1,…
$ ps_ind_03      <dbl> 5, 7, 9, 2, 0, 4, 3, 4, 3, 2, 2, 3, 1, 3, 11, 3…
$ ps_ind_04_cat  <fct> 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1,…
$ ps_ind_05_cat  <fct> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_06_bin  <int> 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_07_bin  <int> 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1,…
$ ps_ind_08_bin  <int> 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0,…
$ ps_ind_09_bin  <int> 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,…
$ ps_ind_10_bin  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_11_bin  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_12_bin  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_13_bin  <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_14      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_15      <dbl> 11, 3, 12, 8, 9, 6, 8, 13, 6, 4, 3, 9, 10, 12, …
$ ps_ind_16_bin  <int> 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0,…
$ ps_ind_17_bin  <int> 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_ind_18_bin  <int> 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1,…
$ ps_reg_01      <dbl> 0.7, 0.8, 0.0, 0.9, 0.7, 0.9, 0.6, 0.7, 0.9, 0.…
$ ps_reg_02      <dbl> 0.2, 0.4, 0.0, 0.2, 0.6, 1.8, 0.1, 0.4, 0.7, 1.…
$ ps_reg_03      <dbl> 0.7180703, 0.7660777, -1.0000000, 0.5809475, 0.…
$ ps_car_01_cat  <fct> 10, 11, 7, 7, 11, 10, 6, 11, 10, 11, 11, 11, 6,…
$ ps_car_02_cat  <fct> 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1,…
$ ps_car_03_cat  <fct> -1, -1, -1, 0, -1, -1, -1, 0, -1, 0, -1, -1, -1…
$ ps_car_04_cat  <fct> 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 8, 0, 0, 0, 0, 9,…
$ ps_car_05_cat  <fct> 1, -1, -1, 1, -1, 0, 1, 0, 1, 0, -1, -1, -1, 1,…
$ ps_car_06_cat  <fct> 4, 11, 14, 11, 14, 14, 11, 11, 14, 14, 13, 11, …
$ ps_car_07_cat  <fct> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,…
$ ps_car_08_cat  <fct> 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0,…
$ ps_car_09_cat  <fct> 0, 2, 2, 3, 2, 0, 0, 2, 0, 2, 2, 0, 2, 2, 2, 0,…
$ ps_car_10_cat  <fct> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,…
$ ps_car_11_cat  <fct> 12, 19, 60, 104, 82, 104, 99, 30, 68, 104, 20, …
$ ps_car_11      <dbl> 2, 3, 1, 1, 3, 2, 2, 3, 3, 2, 3, 3, 3, 3, 1, 2,…
$ ps_car_12      <dbl> 0.4000000, 0.3162278, 0.3162278, 0.3741657, 0.3…
$ ps_car_13      <dbl> 0.8836789, 0.6188165, 0.6415857, 0.5429488, 0.5…
$ ps_car_14      <dbl> 0.3708099, 0.3887158, 0.3472751, 0.2949576, 0.3…
$ ps_car_15      <dbl> 3.605551, 2.449490, 3.316625, 2.000000, 2.00000…
$ ps_calc_01     <dbl> 0.6, 0.3, 0.5, 0.6, 0.4, 0.7, 0.2, 0.1, 0.9, 0.…
$ ps_calc_02     <dbl> 0.5, 0.1, 0.7, 0.9, 0.6, 0.8, 0.6, 0.5, 0.8, 0.…
$ ps_calc_03     <dbl> 0.2, 0.3, 0.1, 0.1, 0.0, 0.4, 0.5, 0.1, 0.6, 0.…
$ ps_calc_04     <dbl> 3, 2, 2, 2, 2, 3, 2, 1, 3, 2, 2, 2, 4, 2, 3, 2,…
$ ps_calc_05     <dbl> 1, 1, 2, 4, 2, 1, 2, 2, 1, 2, 3, 2, 1, 1, 1, 1,…
$ ps_calc_06     <dbl> 10, 9, 9, 7, 6, 8, 8, 7, 7, 8, 8, 8, 8, 10, 8, …
$ ps_calc_07     <dbl> 1, 5, 1, 1, 3, 2, 1, 1, 3, 2, 2, 2, 4, 1, 2, 5,…
$ ps_calc_08     <dbl> 10, 8, 8, 8, 10, 11, 8, 6, 9, 9, 9, 10, 11, 8, …
$ ps_calc_09     <dbl> 1, 1, 2, 4, 2, 3, 3, 1, 4, 1, 4, 1, 1, 3, 3, 2,…
$ ps_calc_10     <dbl> 5, 7, 7, 2, 12, 8, 10, 13, 11, 11, 7, 8, 9, 8, …
$ ps_calc_11     <dbl> 9, 3, 4, 2, 3, 4, 3, 7, 4, 3, 6, 9, 6, 2, 4, 5,…
$ ps_calc_12     <dbl> 1, 1, 2, 2, 1, 2, 0, 1, 2, 5, 3, 2, 3, 0, 1, 2,…
$ ps_calc_13     <dbl> 5, 1, 7, 4, 1, 0, 0, 3, 1, 0, 3, 1, 3, 4, 3, 6,…
$ ps_calc_14     <dbl> 8, 9, 7, 9, 3, 9, 10, 6, 5, 6, 6, 10, 8, 3, 9, …
$ ps_calc_15_bin <int> 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,…
$ ps_calc_16_bin <int> 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1,…
$ ps_calc_17_bin <int> 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1,…
$ ps_calc_18_bin <int> 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,…
$ ps_calc_19_bin <int> 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1,…
$ ps_calc_20_bin <int> 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0,…
```

We split off 25% for validation.

```
# train-test split
id_training <-       sample.int(       nrow    (       porto   )       , size
=       0.75    *       nrow    (       porto   )       )

x_train   <-       porto   [       id_training,]       %>%      select
(       -       target  )
x_test    <-       porto   [       -       id_training,]       %>%
select  (       -       target  )
y_train   <-       porto   [       id_training, "target"]
y_test    <-       porto   [       -       id_training, "target"]
```

The only thing we want to do to the *data* before defining the network is scaling the numeric features. Binary and categorical features can stay as is, with the minor correction that for the categorical ones, we'll actually pass the network the numeric representation of the factor data.

Here is the scaling.

```
train_means <-       colMeans(       x_train [       sapply  (       x_train ,
is.double)      ]       )       %>%      unname  (       )
train_sds <-       apply   (       x_train [       sapply  (       x_train ,
is.double)      ]       , 2       , sd      )       %>%      unname
(       )
train_sds[       train_sds ==       0       ]       <-       0.000001

x_train [       sapply  (       x_train , is.double)      ]       <-
sweep   (
  x_train [       sapply  (       x_train , is.double)      ]       ,
  2       ,
  train_means
  )       %>%
  sweep   (       2       , train_sds, "/"     )
x_test  [       sapply  (       x_test  , is.double)      ]       <-
sweep   (
  x_test  [       sapply  (       x_test  , is.double)      ]       ,
  2       ,
  train_means
  )       %>%
  sweep   (       2       , train_sds, "/"     )
```

When building the network, we need to specify the input and output dimensionalities for the embedding layers. Input dimensionality refers to the number of different symbols that "come in"; in NLP tasks this would be the vocabulary size while here, it's simply the number of values a variable can take. [3] Output dimensionality, the capacity of the internal representation, can then be calculated based on some heuristic. Below, we'll follow a popular rule of thumb that takes the square root of the dimensionality of the input.

So as part one of the network, here we build up the embedding layers in a loop, each wired to the input layer that feeds it:

```
# number of levels per factor, required to specify input dimensionality for
# the embedding layers
n_levels_in <-       map     (       x_train  %>%      select_if(
is.factor)       , compose (       length  , levels  )       )       %>%
  unlist  (       )

# output dimensionality for the embedding layers, need +1 because Python is 0-
       based
n_levels_out <-       n_levels_in %>%      sqrt    (       )       %>%
trunc   (       )       %>%      `+`     (       1       )


# each embedding layer gets its own input layer
```

```
cat_inputs <-       map     (       n_levels_in, function(       l
)       layer_input(       shape =        1       )       )       %>%
  unname   (       )

# construct the embedding layers, connecting each to its input
embedding_layers <-       vector   (       mode =        "list"   , length
=       length   (       cat_inputs)       )
for      (       i       in       1       :       length   (       cat_inputs
)       )       {
  embedding_layer <-        cat_inputs[[      i       ]       ]       %>%
    layer_embedding(       input_dim =        n_levels_in[[      i
]       ]       +       1       , output_dim =       n_levels_out[[
i       ]       ]       )       %>%
    layer_flatten(       )
  embedding_layers[[      i       ]       ]       <-       embedding_layer
}
```

In case you were wondering about the `flatten` layer following each embedding: We need to squeeze out the third dimension (introduced by the embedding layers) from the tensors, effectively rendering them rank-2. That is because we want to combine them with the rank-2 tensor coming out of the dense layer processing the numeric features.

In order to be able to combine it with anything, we have to actually construct that dense layer first. It will be connected to a single input layer, of shape 43, that takes in the numeric features we scaled as well as the binary features we left untouched:

```
# create a single input and a dense layer for the numeric data
quant_input <-       layer_input(       shape =       43      )

quant_dense <-       quant_input %>%      layer_dense(       units =
64      )
```

Are parts assembled, we wire them together using `layer_concatenate`, and we're good to call `keras_model` to create the final graph.

```
intermediate_layers <-       list    (       embedding_layers, list    (
quant_dense)       )       %>%      flatten (       )
inputs    <-       list    (       cat_inputs, list    (       quant_input
)       )       %>%      flatten (       )

l        <-       0.25

output    <-       layer_concatenate(       intermediate_layers)       %>%
  layer_dense(       units =       30       , activation =       "relu"   ,
kernel_regularizer =       regularizer_l2(       l       )       )
%>%
  layer_dropout(       rate =       0.25    )       %>%
  layer_dense(       units =       10       , activation =       "relu"   ,
kernel_regularizer =       regularizer_l2(       l       )       )
%>%
  layer_dropout(       rate =       0.25    )       %>%
  layer_dense(       units =       5       , activation =       "relu"   ,
kernel_regularizer =       regularizer_l2(       l       )       )
%>%
  layer_dropout(       rate =       0.25    )       %>%
  layer_dense(       units =       1       , activation =       "sigmoid",
kernel_regularizer =       regularizer_l2(       l       )       )

model     <-       keras_model(       inputs   , output   )
```

Now, if you've actually read through the whole of this part, you may wish for an easier way to get to this point. So let's switch to feature specs for the rest of this post.

## Feature specs to the rescue

## Feature specs to the rescue

In spirit, the way feature specs are defined follows the example of the recipes package. (It won't make you hungry, though.) You initialize a feature spec with the prediction target – `feature_spec(target ~ .)`, and then use the `%>%` to tell it what to do with individual columns. "What to do" here signifies two things:

- First, how to "read in" the data. Are they numeric or categorical, and if categorical, what am I supposed to do with them? For example, should I treat all distinct symbols as distinct, resulting in, potentially, an enormous count of categories – or should I constrain myself to a fixed number of entities? Or hash them, even?
- Second, optional subsequent transformations. Numeric columns may be bucketized; categorical columns may be embedded. Or features could be combined to capture interaction.

In this post, we demonstrate the use of a subset of `step_` functions. The vignettes on Feature columns and Feature specs illustrate additional functions and their application.

Starting from the beginning again, here is the complete code for data read-in and train-test split in the feature spec version.

```
library ( keras )
library ( tfdatasets)
library ( readr )
library ( dplyr )

path <- "train.csv"

porto <- read_csv ( path )

porto <- porto %>%
  mutate_at( vars ( ends_with( "cat" ) ) ) ,
as.character)

id_training <- sample.int( nrow ( porto ) , size
= 0.75 * nrow ( porto ) )
training <- porto [ id_training,]
testing <- porto [ - id_training,]
```

Data-prep-wise, recall what our goals are: leave alone if binary; scale if numeric; embed if categorical. Specifying all of this does not need more than a few lines of code:

```
ft_spec <- training %>%
  select ( - id ) %>%
  feature_spec( target ~ . ) %>%
  step_numeric_column( ends_with( "bin" ) ) %>%
  step_numeric_column( - ends_with( "bin" ) ,
                       - ends_with( "cat" ) ,
                       normalizer_fn = scaler_standard( )
                       ) %>%
  step_categorical_column_with_vocabulary_list( ends_with( "cat"
) ) %>%
  step_embedding_column( ends_with( "cat" ) ,
                         dimension = function( vocab_size)
as.integer( sqrt ( vocab_size) + 1 )
                       ) %>%
  fit ( )
```

Note how here we are passing in the training set, and just like with `recipes`, we won't need to repeat any of the steps for the validation set. Scaling is taken care of by

`scaler_standard()`, an optional transformation function passed in to `step_numeric_column`. Categorical columns are meant to make use of the complete vocabulary [4] and pipe their outputs into embedding layers.

Now, what actually happened when we called `fit()`? A lot – for us, as we got rid of a ton of manual preparation. For TensorFlow, nothing really – it just came to know about a few pieces in the graph we'll ask it to construct.

But wait, – don't we still have to build up that graph ourselves, connecting and concatenating layers? Concretely, above, we had to:

- create the correct number of input layers, of correct shape; and
- wire them to their matching embedding layers, of correct dimensionality.

So here comes the real magic, and it has two steps.

First, we easily create the input layers by calling `layer_input_from_dataset`:

```
inputs <- layer_input_from_dataset(porto %>% select(-target))
```

And second, we can extract the features from the feature spec and have `layer_dense_features` create the necessary layers based on that information:

```
layer_dense_features(ft_spec$dense_features())
```

Without further ado, we add a few dense layers, and there is our model. Magic!

```
output <- inputs %>%
  layer_dense_features(ft_spec$dense_features()) %>%
  layer_dense(units = 30, activation = "relu",
kernel_regularizer = regularizer_l2(l)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_dense(units = 10, activation = "relu",
kernel_regularizer = regularizer_l2(l)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_dense(units = 5, activation = "relu",
kernel_regularizer = regularizer_l2(l)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_dense(units = 1, activation = "sigmoid",
kernel_regularizer = regularizer_l2(l))

model <- keras_model(inputs, output)
```

How do we feed this model? In the non-feature-columns example, we would have had to feed each input separately, passing a list of tensors. Now we can just pass it the complete training set all at once:

```
model %>% fit(x = training, y = training$target)
```

In the Kaggle competition, submissions are evaluated using the normalized Gini coefficient, which we can calculate with the help of a new metric available in Keras,

`tf$keras$metrics$AUC()`. For training, we can use an approximation to the AUC due to Yan et al. (2003) (Yan et al. 2003). Then training is as straightforward as:

```
auc <- tf$keras$metrics$AUC()

gini <- custom_metric(name = "gini", function(y_true, y_pred) {
  2 * auc(y_true, y_pred) - 1
})

# Yan, L., Dodier, R., Mozer, M. C., & Wolniewicz, R. (2003).
# Optimizing Classifier Performance via an Approximation to the Wilcoxon-Mann-Whitney Statistic.
roc_auc_score <- function(y_true, y_pred) {

  pos = tf$boolean_mask(y_pred, tf$cast(y_true, tf$bool))
  neg = tf$boolean_mask(y_pred, !tf$cast(y_true, tf$bool))

  pos = tf$expand_dims(pos, 0L)
  neg = tf$expand_dims(neg, 1L)

  # original paper suggests performance is robust to exact parameter choice
  gamma = 0.2
  p = 3

  difference = tf$zeros_like(pos * neg) + pos - neg - gamma

  masked = tf$boolean_mask(difference, difference < 0.0)

  tf$reduce_sum(tf$pow(-masked, p))
}

model %>%
  compile(
    loss = roc_auc_score,
    optimizer = optimizer_adam(),
    metrics = list(auc, gini)
  )

model %>%
  fit(
    x = training,
    y = training$target,
    epochs = 50,
    validation_data = list(testing, testing$target),
    batch_size = 512
  )

predictions <- predict(model, testing)
Metrics::auc(testing$target, predictions)
```

After 50 epochs, we achieve an AUC of 0.64 on the validation set, or equivalently, a Gini coefficient of 0.27. Not a bad result for a simple fully connected network!

We've seen how using feature columns automates away a number of steps in setting up the network, so we can spend more time on actually tuning it. This is most impressively demonstrated on a dataset like this, with more than a handful categorical columns. However, to explain a bit more what to pay attention to when using feature columns, it's better to

choose a smaller example where we can easily do some peeking around.

Let's move on to the second application.

# Interactions, and what to look out for

To demonstrate the use of `step_crossed_column` to capture interactions, we make use of the `rugged` dataset from Richard McElreath's *rethinking* package.

We want to predict log GDP based on terrain ruggedness, for a number of countries (170, to be precise). However, the effect of ruggedness is different in Africa as opposed to other continents. Citing from *Statistical Rethinking*

> *It makes sense that ruggedness is associated with poorer countries, in most of the world. Rugged terrain means transport is difficult. Which means market access is hampered. Which means reduced gross domestic product. So the reversed relationship within Africa is puzzling. Why should difficult terrain be associated with higher GDP per capita?*

> *If this relationship is at all causal, it may be because rugged regions of Africa were protected against the Atlantic and Indian Ocean slave trades. Slavers preferred to raid easily accessed settlements, with easy routes to the sea. Those regions that suffered under the slave trade understandably continue to suffer economically, long after the decline of slave-trading markets. However, an outcome like GDP has many influences, and is furthermore a strange measure of economic activity. So it is hard to be sure what's going on here.*

While the causal situation is difficult, the purely technical one is easily described: We want to learn an interaction. We could rely on the network finding out by itself (in this case it probably will, if we just give it enough parameters). But it's an excellent occasion to showcase the new `step_crossed_column`.

Loading the dataset, zooming in on the variables of interest, and normalizing them the way it is done in *Rethinking*, we have:

```
library (       rethinking)
library (       keras    )
library (       tfdatasets)
library (       tibble   )

data    (       rugged   )

d        <-       rugged
d        <-       d       [       complete.cases(       d       $
rgdppc_2000)       , ]

df       <-       tibble   (
  log_gdp =        log     (       d       $       rgdppc_2000)       /
mean    (       log     (       d       $       rgdppc_2000)       )       ,
  rugged =        d       $       rugged  /       max     (       d
$       rugged  )       ,
  africa =        d       $       cont_africa
)

df       %>%      glimpse (       )
```

```
Observations: 170
```

```
Variables: 3
$ log_gdp <dbl> 0.8797119, 0.9647547, 1.1662705, 1.1044854, 0.9149038,…
$ rugged  <dbl> 0.1383424702, 0.5525636891, 0.1239922606, 0.1249596904…
$ africa  <int> 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, …
```

Now, let's first forget about the interaction and do the very minimal thing required to work with this data. `rugged` should be a numeric column, while `africa` is categorical in nature, which means we use one of the `step_categorical_[...]` functions on it. (In this case we happen to know there are just two categories, Africa and not-Africa, so we could as well treat the column as numeric like in the previous example; but in other applications that won't be the case, so here we show a method that generalizes to categorical features in general.)

So we start out creating a feature spec and adding the two predictor columns. We check the result using `feature_spec`'s `dense_features()` method:

```
ft_spec  <-       training %>%
  feature_spec(       log_gdp  ~        .       )        %>%
  step_numeric_column(       rugged  )        %>%
  step_categorical_column_with_identity(       africa   , num_buckets =
2       )
  fit     (       )

ft_spec $       dense_features(       )
```

```
$rugged
NumericColumn(key='rugged', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None)
```

Hm, that doesn't look too good. Where'd `africa` go? In fact, there is one more thing we should have done: convert the categorical column to an *indicator column*. Why?

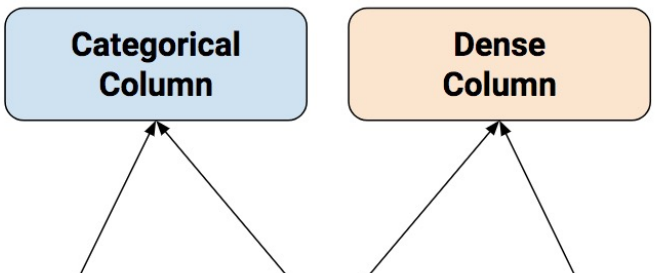
The rule of thumb is, whenever you have something *categorical*, including *crossed*, you need to then transform it into something *numeric*, which includes *indicator* and *embedding*.

Being a heuristic, this rule works overall, and it matches our intuition. There's one exception though, `step_bucketized_column`, which although it "feels" categorical actually does not need that conversion.

Therefore, it is best to supplement that intuition with a simple lookup diagram, which is also part of the feature columns vignette.

With this diagram, the simple rule is: *We always need to end up with something that inherits from `DenseColumn`*. So:

- `step_numeric_column`, `step_indicator_column`, and `step_embedding_column` are standalone;
- `step_bucketized_column` is, too, however categorical it "feels"; and
- all `step_categorical_column_[...]`, as well as `step_crossed_column`, need to be transformed using one the *dense* column types.



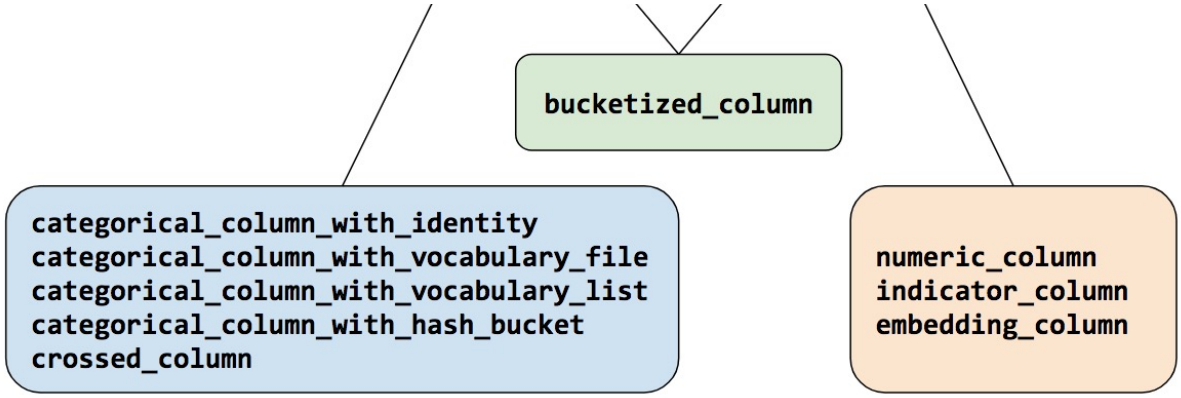



Figure 1: For use with Keras, all features need to end up inheriting from DenseColumn somehow.

Thus, we can fix the situation like so:

```
ft_spec  <-       training %>%
  feature_spec(       log_gdp  ~        .       )       %>%
  step_numeric_column(       rugged  )       %>%
  step_categorical_column_with_identity(       africa  , num_buckets =
2       )       %>%
  step_indicator_column(       africa  )       %>%
  fit     (       )
```

and now `ft_spec$dense_features()` will show us

```
$rugged
NumericColumn(key='rugged', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None)

$indicator_africa
IndicatorColumn(categorical_column=IdentityCategoricalColumn(key='africa', number_buckets=2.0, default_value=N
```

What we really wanted to do is capture the interaction between ruggedness and continent. To this end, we first *bucketize* `rugged`, and then cross it with – already binary – `africa`. As per the rules, we finally transform into an *indicator* column:

```
ft_spec  <-       training %>%
  feature_spec(       log_gdp  ~        .       )       %>%
  step_numeric_column(       rugged  )       %>%
  step_categorical_column_with_identity(       africa  , num_buckets =
2       )       %>%
  step_indicator_column(       africa  )       %>%
  step_bucketized_column(       rugged  ,
                        boundaries =       c       (       0.1     , 0.2
, 0.3     , 0.4     , 0.5     , 0.6     , 0.8     )       )       %>%
  step_crossed_column(       africa_rugged_interact =       c       (
africa  , bucketized_rugged)       ,
                       hash_bucket_size =       16      )       %>%
  step_indicator_column(       africa_rugged_interact)       %>%
  fit     (       )
```

Looking at this code you may be asking yourself, now how many features do I have in the model? Let's check.

```
ft_spec $       dense_features(       )
```

```
$rugged
NumericColumn(key='rugged', shape=(1,), default_value=None, dtype=tf.float32, normalizer_fn=None)


$indicator_africa
IndicatorColumn(categorical_column=IdentityCategoricalColumn(key='africa', number_buckets=2.0, default_value=N
```

```
$bucketized_rugged
BucketizedColumn(source_column=NumericColumn(key='rugged', shape=(1,), default_value=None, dtype=tf.float32, r

$indicator_africa_rugged_interact
IndicatorColumn(categorical_column=CrossedColumn(keys=(IdentityCategoricalColumn(key='africa', number_buckets=
```

We see that all features, original or transformed, are kept, as long as they inherit from `DenseColumn`. This means that, for example, the non-bucketized, continuous values of `rugged` are used as well.

Now setting up the training goes as expected.

```
inputs <- layer_input_from_dataset( df %>% select
( - log_gdp ) )

output <- inputs %>%
  layer_dense_features( ft_spec $ dense_features( )
) %>%
  layer_dense( units = 8 , activation = "relu"
) %>%
  layer_dense( units = 8 , activation = "relu"
) %>%
  layer_dense( units = 1 )

model <- keras_model( inputs , output )

model %>% compile ( loss = "mse" , optimizer =
"adam" , metrics = "mse" )

history <- model %>% fit (
  x = training ,
  y = training $ log_gdp ,
  validation_data = list ( testing , testing $ log_gdp
) ,
  epochs = 100 )
```

Just as a sanity check, the final loss on the validation set for this code was ~ 0.014. But really this example did serve different purposes.

# In a nutshell

*Feature specs* are a convenient, elegant way of making categorical data available to Keras, as well as to chain useful transformations like bucketizing and creating crossed columns. The time you save data wrangling may go into tuning and experimentation. Enjoy, and thanks for reading!

RStudio AI Blog: Introductory time ...
a year ago • 7 comments
This post is an introduction to time-series forecasting with torch. Central topics …

RStudio AI Blog: Time series prediction ...
2 years ago • 8 comments
In a recent post, we showed how an LSTM autoencoder, regularized by false …

RStudio AI Blog: torch time series ...
10 months ago • 2 comments
We continue our exploration of time-series forecasting with torch, moving on to …

RStudi Getting
a year ag
In this fir four-part present t



## Footnotes

1. see e.g. Entity embeddings for fun and profit [↩]
2. Having mentioned, above, that these really may be continuous or ordinal, from now on we'll just call them numeric as we won't make further use of the distinction. However, fitting ordinal data with neural networks is definitely a topic that would deserve its own post. [↩]
3. In this dataset, every categorical variable is a column of singleton integers. [↩]
4. as indicated by leaving out the optional `vocabulary_list` [↩]

## References

Yan, Lian, Robert H Dodier, Michael Mozer, and Richard H Wolniewicz. 2003. "Optimizing Classifier Performance via an Approximation to the Wilcoxon-Mann-Whitney Statistic." In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 848–55.

## Reuse



## Citation

For attribution, please cite this work as

```
Falbel & Keydana (2019, July 9). RStudio AI Blog: TensorFlow feature columns: Transforming your data
recipes-style. Retrieved from https://blogs.rstudio.com/tensorflow/posts/2019-07-09-feature-columns/
```

BibTeX citation

```
@misc{falbelkeydana2019featurecols,
  author = {Falbel, Daniel and Keydana, Sigrid},
  title = {RStudio AI Blog: TensorFlow feature columns: Transforming your data recipes-style},
  url = {https://blogs.rstudio.com/tensorflow/posts/2019-07-09-feature-columns/},
  year = {2019}
}
```