**Overview**

The kerasformula package offers a high-level interface for the R interface to Keras. It’s main interface is the kms function, a regression-style interface to keras\_model\_sequential that uses formulas and sparse matrices.

The kerasformula package is available on CRAN, and can be installed with:

# install the kerasformula package

install.packages("kerasformula") # or devtools::install\_github("rdrr1990/kerasformula")

library(kerasformula)

# install the core keras library (if you haven't already done so)

# see ?install\_keras() for options e.g. install\_keras(tensorflow = "gpu")

install\_keras()

**The kms() function**

Many classic machine learning tutorials assume that data come in a relatively homogenous form (e.g., pixels for digit recognition or word counts or ranks) which can make coding somewhat cumbersome when data is contained in a heterogenous data frame. kms() takes advantage of the flexibility of R formulas to smooth this process.

KMS Process

# NOT RUN {

if(is\_keras\_available()){

mtcars$make <- unlist(lapply(strsplit(rownames(mtcars), " "), function(tokens) tokens[1]))

company <- kms(make ~ ., mtcars, Nepochs = 1, verbose=0)

# out of sample accuracy

pCorrect <- mean(company$y\_test == company$predictions)

pCorrect

company$confusion

# plot(history$company) # helps pick Nepochs

# below

# find the default settings for layers

company <- kms(make ~ ., mtcars,

units = c(256, 128),

activation = c("relu", "relu", "softmax"),

dropout = 0.4,

use\_bias = TRUE,

kernel\_initializer = NULL,

kernel\_regularizer = "regularizer\_l1",

bias\_regularizer = "regularizer\_l1",

activity\_regularizer = "regularizer\_l1",

Nepochs = 1, verbose=0

)

# ?predict.kms\_fit to see how to predict on newdata

}else{

cat("Please run install\_keras() before using kms(). ?install\_keras for options like gpu.")

}

# }

kms builds dense neural nets and, after fitting them, returns a single object with predictions, measures of fit, and details about the function call. kms accepts a number of parameters including the loss and activation functions found in keras. kms also accepts compiled keras\_model\_sequential objects allowing for even further customization. This little demo shows how kms can aid is model building and hyperparameter selection (e.g., batch size) starting with raw data gathered using library(rtweet).

Let’s look at #rstats tweets (excluding retweets) for a six-day period ending January 24, 2018 at 10:40. This happens to give us a nice reasonable number of observations to work with in terms of runtime (and the purpose of this document is to show syntax, not build particularly predictive models).

rstats <- search\_tweets("#rstats", n = 10000, include\_rts = FALSE)

dim(rstats)

[1] 2840 42

Suppose our goal is to predict how popular tweets will be based on how often the tweet was retweeted and favorited (which correlate strongly).

cor(rstats$favorite\_count, rstats$retweet\_count, method="spearman")

[1] 0.7051952

Since few tweeets go viral, the data are quite skewed towards zero.

**Getting the most out of formulas**

Let’s suppose we are interested in putting tweets into categories based on popularity but we’re not sure how finely-grained we want to make distinctions. Some of the data, like rstats$mentions\_screen\_name comes in a list of varying lengths, so let’s write a helper function to count non-NA entries.

n <- function(x) {

unlist(lapply(x, function(y){length(y) - is.na(y[1])}))

}

Let’s start with a dense neural net, the default of kms. We can use base R functions to help clean the data–in this case, cut to discretize the outcome, grepl to look for key words, and weekdays and format to capture different aspects of the time the tweet was posted.

breaks <- c(-1, 0, 1, 10, 100, 1000, 10000)

popularity <- kms(cut(retweet\_count + favorite\_count, breaks) ~ screen\_name + source +

n(hashtags) + n(mentions\_screen\_name) +

n(urls\_url) + nchar(text) +

grepl('photo', media\_type) +

weekdays(created\_at) +

format(created\_at, '%H'), rstats)

plot(popularity$history) + ggtitle(paste("#rstat popularity:",

paste0(round(100\*popularity$evaluations$acc, 1), "%"),

"out-of-sample accuracy")) + theme\_minimal()

popularity$confusion

popularity$confusion

(-1,0] (0,1] (1,10] (10,100] (100,1e+03] (1e+03,1e+04]

(-1,0] 37 12 28 2 0 0

(0,1] 14 19 72 1 0 0

(1,10] 6 11 187 30 0 0

(10,100] 1 3 54 68 0 0

(100,1e+03] 0 0 4 10 0 0

(1e+03,1e+04] 0 0 0 1 0 0

The model only classifies about 55% of the out-of-sample data correctly and that predictive accuracy doesn’t improve after the first ten epochs. The confusion matrix suggests that model does best with tweets that are retweeted a handful of times but overpredicts the 1-10 level. The history plot also suggests that out-of-sample accuracy is not very stable. We can easily change the breakpoints and number of epochs.

breaks <- c(-1, 0, 1, 25, 50, 75, 100, 500, 1000, 10000)

popularity <- kms(cut(retweet\_count + favorite\_count, breaks) ~

n(hashtags) + n(mentions\_screen\_name) + n(urls\_url) +

nchar(text) +

screen\_name + source +

grepl('photo', media\_type) +

weekdays(created\_at) +

format(created\_at, '%H'), rstats, Nepochs = 10)

plot(popularity$history) + ggtitle(paste("#rstat popularity (new breakpoints):",

paste0(round(100\*popularity$evaluations$acc, 1), "%"),

"out-of-sample accuracy")) + theme\_minimal()

That helped some (about 5% additional predictive accuracy). Suppose we want to add a little more data. Let’s first store the input formula.

pop\_input <- "cut(retweet\_count + favorite\_count, breaks) ~

n(hashtags) + n(mentions\_screen\_name) + n(urls\_url) +

nchar(text) +

screen\_name + source +

grepl('photo', media\_type) +

weekdays(created\_at) +

format(created\_at, '%H')"

Here we use paste0 to add to the formula by looping over user IDs adding something like:

grepl("12233344455556", mentions\_user\_id)

mentions <- unlist(rstats$mentions\_user\_id)

mentions <- unique(mentions[which(table(mentions) > 5)]) # remove infrequent mentions

mentions <- mentions[!is.na(mentions)] # drop NA

for(i in mentions)

pop\_input <- paste0(pop\_input, " + ", "grepl(", i, ", mentions\_user\_id)")

popularity <- kms(pop\_input, rstats)

That helped a touch but the predictive accuracy is still fairly unstable across epochs…

**Customizing layers with kms()**

We could add more data, perhaps add individual words from the text or some other summary stat (mean(text %in% LETTERS) to see if all caps explains popularity). But let’s alter the neural net.

The input.formula is used to create a sparse model matrix. For example, rstats$source (Twitter or Twitter-client application type) and rstats$screen\_name are character vectors that will be dummied out. How many columns does it have?

popularity$P

[1] 1277

Say we wanted to reshape the layers to transition more gradually from the input shape to the output.

popularity <- kms(pop\_input, rstats,

layers = list(units = c(1024, 512, 256, 128, NA),

activation = c("relu", "relu", "relu", "relu", "softmax"),

dropout = c(0.5, 0.45, 0.4, 0.35, NA)))

kms builds a keras\_sequential\_model(), which is a stack of linear layers. The input shape is determined by the dimensionality of the model matrix (popularity$P) but after that users are free to determine the number of layers and so on. The kms argument layers expects a list, the first entry of which is a vector units with which to call [keras::layer\_dense()](https://tensorflow.rstudio.com/keras/reference/layer_dense.html). The first element the number of units in the first layer, the second element for the second layer, and so on (NA as the final element connotes to auto-detect the final number of units based on the observed number of outcomes). activation is also passed to layer\_dense() and may take values such as softmax, relu, elu, and linear. (kms also has a separate parameter to control the optimizer; by default kms(... optimizer = 'rms\_prop').) The dropout that follows each dense layer rate prevents overfitting (but of course isn’t applicable to the final layer).

Code Chunks – Keras:: Layer\_Dense()

Add a densely-connected NN layer to an output

* [Arguments](https://tensorflow.rstudio.com/reference/keras/layer_dense/#arguments)
* [Input and Output Shapes](https://tensorflow.rstudio.com/reference/keras/layer_dense/#input-and-output-shapes)
* [See also](https://tensorflow.rstudio.com/reference/keras/layer_dense/#see-also)

Implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use\_bias is TRUE). Note: if the input to the layer has a rank greater than 2, then it is flattened prior to the initial dot product with kernel.

layer\_dense(

object,

units,

activation = NULL,

use\_bias = TRUE,

kernel\_initializer = "glorot\_uniform",

bias\_initializer = "zeros",

kernel\_regularizer = NULL,

bias\_regularizer = NULL,

activity\_regularizer = NULL,

kernel\_constraint = NULL,

bias\_constraint = NULL,

input\_shape = NULL,

batch\_input\_shape = NULL,

batch\_size = NULL,

dtype = NULL,

name = NULL,

trainable = NULL,

weights = NULL

)

Arguments

|  |  |
| --- | --- |
| object | Model or layer object |
| units | Positive integer, dimensionality of the output space. |
| activation | Name of activation function to use. If you don't specify anything, no activation is applied (ie. "linear" activation: a(x) = x). |
| use\_bias | Whether the layer uses a bias vector. |
| kernel\_initializer | Initializer for the kernel weights matrix. |
| bias\_initializer | Initializer for the bias vector. |
| kernel\_regularizer | Regularizer function applied to the kernel weights matrix. |
| bias\_regularizer | Regularizer function applied to the bias vector. |
| activity\_regularizer | Regularizer function applied to the output of the layer (its "activation").. |
| kernel\_constraint | Constraint function applied to the kernel weights matrix. |
| bias\_constraint | Constraint function applied to the bias vector. |
| input\_shape | Dimensionality of the input (integer) not including the samples axis. This argument is required when using this layer as the first layer in a model. |
| batch\_input\_shape | Shapes, including the batch size. For instance, batch\_input\_shape=c(10, 32) indicates that the expected input will be batches of 10 32-dimensional vectors. batch\_input\_shape=list(NULL, 32) indicates batches of an arbitrary number of 32-dimensional vectors. |
| batch\_size | Fixed batch size for layer |
| dtype | The data type expected by the input, as a string (float32, float64, int32...) |
| name | An optional name string for the layer. Should be unique in a model (do not reuse the same name twice). It will be autogenerated if it isn't provided. |
| trainable | Whether the layer weights will be updated during training. |
| weights | Initial weights for layer. |

Input and Output Shapes

Input shape: nD tensor with shape: (batch\_size, ..., input\_dim). The most common situation would be a 2D input with shape (batch\_size, input\_dim).

Output shape: nD tensor with shape: (batch\_size, ..., units). For instance, for a 2D input with shape (batch\_size, input\_dim), the output would have shape (batch\_size, unit).

**Choosing a Batch Size**

By default, kms uses batches of 32. Suppose we were happy with our model but didn’t have any particular intuition about what the size should be.

Nbatch <- c(16, 32, 64)

Nruns <- 4

accuracy <- matrix(nrow = Nruns, ncol = length(Nbatch))

colnames(accuracy) <- paste0("Nbatch\_", Nbatch)

est <- list()

for(i in 1:Nruns){

for(j in 1:length(Nbatch)){

est[[i]] <- kms(pop\_input, rstats, Nepochs = 2, batch\_size = Nbatch[j])

accuracy[i,j] <- est[[i]][["evaluations"]][["acc"]]

}

}

colMeans(accuracy)

Nbatch\_16 Nbatch\_32 Nbatch\_64

0.5088407 0.3820850 0.5556952

For the sake of curtailing runtime, the number of epochs was set arbitrarily short but, from those results, 64 is the best batch size.

**Making predictions for new data**

Thus far, we have been using the default settings for kms which first splits data into 80% training and 20% testing. Of the 80% training, a certain portion is set aside for validation and that’s what produces the epoch-by-epoch graphs of loss and accuracy. The 20% is only used at the end to assess predictive accuracy. But suppose you wanted to make predictions on a new data set…

popularity <- kms(pop\_input, rstats[1:1000,])

predictions <- predict(popularity, rstats[1001:2000,])

predictions$accuracy

[1] 0.579

# predictions$confusion

Because the formula creates a dummy variable for each screen name and mention, any given set of tweets is all but guaranteed to have different columns. predict.kms\_fit is an S3 method that takes the new data and constructs a (sparse) model matrix that preserves the original structure of the training matrix. predict then returns the predictions along with a confusion matrix and accuracy score.

If your newdata has the same observed levels of y and columns of x\_train (the model matrix), you can also use keras::predict\_classes on object$model.

Generates probability or class probability predictions for the input samples.

predict\_proba(object, x, batch\_size = NULL, verbose = 0, steps = NULL)

predict\_classes(object, x, batch\_size = NULL, verbose = 0, steps = NULL)

## Arguments

|  |  |
| --- | --- |
| object | Keras model object |
| x | Input data (vector, matrix, or array) |
| batch\_size | Integer. If unspecified, it will default to 32. |
| verbose | Verbosity mode, 0 or 1. |
| steps | Total number of steps (batches of samples) before declaring the evaluation round finished. The default NULL is equal to the number of samples in your dataset divided by the batch size. |

**Using a compiled Keras model**

This section shows how to input a model compiled in the fashion typical to library(keras), which is useful for more advanced models. Here is an example for lstm analogous to the imbd with Keras example.

k <- keras\_model\_sequential()

k %>%

layer\_embedding(input\_dim = popularity$P, output\_dim = popularity$P) %>%

layer\_lstm(units = 512, dropout = 0.4, recurrent\_dropout = 0.2) %>%

layer\_dense(units = 256, activation = "relu") %>%

layer\_dropout(0.3) %>%

layer\_dense(units = 8, # number of levels observed on y (outcome)

activation = 'sigmoid')

k %>% compile(

loss = 'categorical\_crossentropy',

optimizer = 'rmsprop',

metrics = c('accuracy')

)

popularity\_lstm <- kms(pop\_input, rstats, k)