R Lab: Data Pre-Processing for Keras and TensorFlow

In this lab, we discuss aspects of data pre-processing needed to get raw data into a format required by Keras and TensorFlow. In particular, they require numerical datasets that are stored as tensors (i.e., an array in R). To start, we pre-process the categorical variables in the dataset, followed by the numerical variables.

The deliverable for this lab is a Word or PDF file containing responses to the exercises at the end of the lab. The deliverable should be submitted through Canvas.

Data Pre-Processing

We first load any R libraries that will be used. Note that Keras and TensorFlow are not needed in this lab, since we focus on the pre-processing steps performed before using Keras and TensorFlow functions. Therefore, we do not need our Python virtual environment.

```r
library(dplyr)
library(caret)
```

The dataset we are using can be found in the the same location in Canvas as this lab. An observation (row) in this dataset corresponds to measurements and characteristics of a 30x30 square meter area of land, as well as whether or not that area of land has Lodgepole Pine as its forest cover. We wish to identify areas with Lodgepole Pine as its forest cover, since this type of tree is useful for certain construction projects.

Categorical Features

There are 10 numerical variables and 2 categorical variables (not including the ‘lodgepole_pine’ target), so the data pre-processing we need to perform involves numerical encoding these 2 categorical variables, which are ‘wilderness_area’, and ‘soil_type’.

First, place the ‘lab_3_data.csv’ file into the working directory and load the dataset into R. The dataset is then divided into two separate datasets: the dataset that will be used to train the models and the dataset that will be used to test the models. These datasets are called the training and test sets, respectively. The reasoning for dividing the entire dataset into these two separate datasets will become clearer in the following weeks as they are discussed in more detail. For now, it is enough to see that we do not want to evaluate our models on the same data that was used to train them, since this would not give us any insight into how the models perform on new observations not yet seen. We use a 75%/25% split for dividing the entire dataset into these two datasets.

```r
data <- read.csv("lab_3_data.csv")
training_ind <- createDataPartition(data$lodgepole_pine,
                                    p = 0.75,
                                    list = FALSE,
                                    times = 1)
training_set <- data[training_ind, ]
test_set <- data[-training_ind, ]
```

Once we divide the dataset into training and test sets, we only use the training set to build the models. In particular, when we pre-process the test set, everything should be based on how we pre-processed the training set. This will become clearer when we scale the numerical variables later.

The categorical variables ‘wilderness_area’ and ‘soil_type’ are initially imported into R as character-valued, so we need to manually convert them into R factor variables. However, if we want to group any values of these variables together, we should perform these groupings while the variables are still character-valued. To decide if any values should be grouped together, we first look at their unique values.

```
unique(training_set$wilderness_area)
```

```
## [1] "wilderness_area_1" "wilderness_area_3" "wilderness_area_4"
## [4] "wilderness_area_2"
```

```
unique(training_set$soil_type)
```

```
##  [1] "soil_type_30" "soil_type_12" "soil_type_29" "soil_type_20" "soil_type_23"
##  [6] "soil_type_24" "soil_type_22" "soil_type_32" "soil_type_10" "soil_type_11"
## [11] "soil_type_5"  "soil_type_33" "soil_type_17" "soil_type_13" "soil_type_31"
## [16] "soil_type_2"  "soil_type_1"  "soil_type_3"  "soil_type_6"  "soil_type_39"
## [21] "soil_type_14" "soil_type_40" "soil_type_4"  "soil_type_38" "soil_type_35"
## [26] "soil_type_27" "soil_type_19" "soil_type_16" "soil_type_18" "soil_type_8"
## [31] "soil_type_9"  "soil_type_28" "soil_type_34" "soil_type_37" "soil_type_21"
## [36] "soil_type_36" "soil_type_26" "soil_type_25"
```

There are only 4 unique values of the ‘wilderness_area’ variable, so it doesn’t seem necessary to group them. However, the ‘soil_type’ variable has several unique values, so one-hot encoding will lead to several features. In an effort to reduce the number of features we get from one-hot encoding ‘soil_type’, we find the top 20 values and group the rest into an ‘other’ category.

```
top_20_soil_types <- training_set %>%
  group_by(soil_type) %>%
  summarise(count = n()) %>%
  arrange(desc(count)) %>%
  select(soil_type) %>%
  top_n(20)

training_set$soil_type <- ifelse(training_set$soil_type %in% top_20_soil_types$soil_type,
                                 training_set$soil_type,
                                 "other")
```

Next, we convert the ‘wilderness_area’ and ‘soil_type’ variables into factors.

```
training_set$wilderness_area <- factor(training_set$wilderness_area)
training_set$soil_type <- factor(training_set$soil_type)
```

We can tell if R is now treating ‘wilderness_area’ and ‘soil_type’ as categorical variables using the ‘class’ function. In fact, it is a good idea to check the class of every column, to make sure R is treating each appropriately.

```
class(training_set$wilderness_area)
```

```
## [1] "factor"
```

```
class(training_set$soil_type)
```

```
## [1] "factor"
```

Categorical variables are called factors in R, so R is indeed treating them correctly. We should also be able to see the levels of the categorical variables using the ‘levels’ function.

```
levels(training_set$wilderness_area)
```

```
## [1] "wilderness_area_1" "wilderness_area_2" "wilderness_area_3"
## [4] "wilderness_area_4"
```

```
levels(training_set$soil_type)
```

```
##  [1] "other"        "soil_type_27" "soil_type_28" "soil_type_29" "soil_type_3"
##  [6] "soil_type_30" "soil_type_31" "soil_type_32" "soil_type_33" "soil_type_34"
## [11] "soil_type_35" "soil_type_36" "soil_type_37" "soil_type_38" "soil_type_39"
## [16] "soil_type_4"  "soil_type_40" "soil_type_5"  "soil_type_6"  "soil_type_8"
## [21] "soil_type_9"
```

It is important to ensure that R recognizes them as categorical variables, since the R function we use for one-hot encoding ('dummyVars', explained next) expects the categorical variables to be factors.

One-Hot Encoding the Training Set

To implement one-hot encoding of the categorical variables, we use the 'dummyVars' function in the 'caret' R library. The 'dummyVars' function takes categorical variables as inputs and outputs an encoder that will know how to one-hot encode the categorical variables for any dataset containing them. The 'levelsOnly' argument specifies whether we want the names of the one-hot encoded features to be the names of the original columns appended to the levels or just the levels only. Also, the 'fullRank' argument is used to specify whether we want the resulting one-hot encoded features to be of full rank; if 'fullRank' is set to 'FALSE' (the default), then all levels from each categorical variable will get a corresponding one-hot encoded feature, if 'fullRank' is set to 'TRUE', then all but one level from each categorical variable will get a corresponding one-hot encoded feature. It is important to set 'fullRank' to 'TRUE' if one of the machine learning models that will be used requires full rank, such as linear regression.

We use the 'dummyVars' function on the training set only, since we treat the test set as not available for any feature engineering or model building. Thus, the test set is truly held-out and can be used for model evaluation, as desired. We might encounter the situation where a level appears in the test set but not the training set. To resolve this situation, a catch-all level, such as 'other', can be added to the training set as was done earlier.

```
onehot_encoder <- dummyVars(~ wilderness_area + soil_type,
                            training_set[, c("wilderness_area", "soil_type")],
                            levelsOnly = TRUE,
                            fullRank = TRUE)
```

Once the encoder is created, we can apply it to any dataset containing the categorical variables. We first apply it to the training set.

```
onehot_enc_training <- predict(onehot_encoder,
                               training_set[, c("wilderness_area", "soil_type")])
```

The data frame 'onehot_enc_training' holds the one-hot encodings of the two categorical variables for the training set, so we can combine this data frame with the training set.

```
training_set <- cbind(training_set, onehot_enc_training)
```

The training set now contains the one-hot encoded features and can now be used in a machine learning algorithm (e.g., random forest).

One-Hot Encoding the Test Set

We perform the same steps on the test set that were used to process the training set. For one-hot encoding of the test set, we use the encoder that was built on the training set.

```
test_set$soil_type <- ifelse(test_set$soil_type %in% top_20_soil_types$soil_type,
                             test_set$soil_type,
                             "other")
```

```r
test_set$wilderness_area <- factor(test_set$wilderness_area)
test_set$soil_type <- factor(test_set$soil_type)

onehot_enc_test <- predict(onehot_encoder, test_set[, c("wilderness_area", "soil_type")])
test_set <- cbind(test_set, onehot_enc_test)
```

Note, for one-hot encoding of categorical variables that will be used with machine learning methods relying on full-rank parameterizations, e.g., linear regression, the 'fullRank' argument of the 'dummyVars' function should be set to TRUE. This will drop one one-hot encoded column for each categorical variable, since for a categorical variable with n levels, knowing whether an observation is one of the first $n-1$ levels tells us whether or not it is the last level. Using a less than full-rank parameterization will make methods like linear regression numerically unstable.

Numerical Features

Numerical variables are more straightforward to work with, since they can typically be used as-is in a machine learning algorithm. However, there is one very important recommendation: numerical variables should be scaled. It is most critical to scale them when using machine learning methods that rely on gradient-descent for optimization (such as neural networks) or a notion of distance (such as k-means clustering, principal component analysis, or k-nearest neighbor). The reason for the latter is to prevent one numerical variable from dominating the others simply because they are measured in different units (e.g., meters vs miles).

There are several ways we can scale the numerical variables, but the most common is to scale each numerical variable to have a mean of zero and a standard deviation of one. This can be achieved with the 'scale' function. Note, we also scale the one-hot encoded variables. We exclude the original categorical variables 'wilderness_area' and 'soil_type' in columns 11 and 12, respectively. We also exclude the target variable 'lodgepole_pine' in column 13, since we do not want to scale the target variable: it needs to be 0/1 valued!

```r
test_set[, -c(11:13)] <- scale(test_set[, -c(11:13)],
                        center = apply(training_set[, -c(11:13)], 2, mean),
                        scale = apply(training_set[, -c(11:13)], 2, sd))
training_set[, -c(11:13)] <- scale(training_set[, -c(11:13)])
```

In the code above, we first apply the scaling to the test set, using the means and standard deviations of each variable from the training set as the center and scale, respectively. Again, all of the pre-processing is based on the training set. Here, the scaling of the test set is performed first, so that we can calculate the means and standard deviations from the training set before we perform the scaling on the training set. Since the scaling has now been performed on all numerical variables, we can be sure that each of the scaled variables is in the same scale.

At this point, both the training set and test set only contain numerical values (not including the original 'wilderness_area' and 'soil_type' variables, which will be removed), so they are ready to be converted into tensors. This is done using the 'array' function in R. Note: no pre-processing was needed for the target variable 'lodgepole_pine', since it was already 0/1 valued. If not, then the target needs to be pre-processed so that it is only 0/1 valued, where 1 represents the positive class and 0 represents the negative class.

```r
training_features <- array(data = unlist(training_set[, -c(11:13)]),
                    dim = c(nrow(training_set), 33))
training_labels <- array(data = unlist(training_set[, 13]),
                    dim = c(nrow(training_set)))

test_features <- array(data = unlist(test_set[, -c(11:13)]),
                    dim = c(nrow(test_set), 33))
test_labels <- array(data = unlist(test_set[, 13]),
                    dim = c(nrow(test_set)))
```

Now, we are ready to use Keras and TensorFlow to build a dense feed-forward neural network on the training set and evaluate it on the test set. This will be discussed in the next lab.

Exercises

1) After working through all of the code in the lab, run 'head(training_features)' and 'head(test_features)'. Copy and paste the output.
2) What is the rank of the tensor 'training_features'? What is the shape of 'training_features'? How many dimensions does 'training_features' have along the second axis?
3) State two situations where scaling the numerical variables is important.