Determining Alpha

Spencer Matthews

3/1/2021

Introduction and Brief Overview

Creating the Data and Models

First we will set up our python environment with the modules and functions that we will need. In this process, we will need numpy and pandas, as well as shap to eventually get the SHAP values for our models. We will be using sickit-learn gradient-boosted trees for our models, so we import the function needed for that as well.

```
import numpy as np
import pandas as pd
import shap
from sklearn.ensemble import GradientBoostingRegressor
np.random.seed(15)
```

We will then create our data, using random uniform distributions on various scales. There will be three different responses, where the first two are simply additive and the last one is the first two multiplied together. This will simulate a simple situation of a two-part model where we multiply the outputs, as we will be doing with the CAS data.

```
# the covariates
x1 = np.random.uniform(low = -10, high = 10, size = 1000)
x2 = np.random.uniform(low = 0, high = 20, size = 1000)
x3 = np.random.uniform(low = -5, high = -1, size = 1000)

dat = {"x1":x1, "x2":x2, "x3":x3}
X = pd.DataFrame(data = dat)

# the first set of responses
y1 = x1 + x2 + x3
y2 = 2*x1 + 2*x2 + 4*x3
y3 = y1 * y2
```

Finally, we will fit the models with minimal tuning parameters and calculate the SHAP values. It will also be important to have the explainer itself so we can get the expected value into R. Also, we will want to calculate what our multiplicative model will predict, so we can verify that the additive property of SHAP values still holds up. And finally, it is a good idea to save the third model prediction so that we know how that model does in comparison to our multiplicative model.

```
# Get the models
mody1 = GradientBoostingRegressor(loss = "ls", min_samples_leaf = 2)
mody1.fit(X, y1)
```

```
## GradientBoostingRegressor(min_samples_leaf=2)
```

```
mody2 = GradientBoostingRegressor(loss = "ls", min_samples_leaf = 2)
mody2.fit(X, y2)
```

```
## GradientBoostingRegressor(min_samples_leaf=2)
```

```
mody3 = GradientBoostingRegressor(loss = "ls", min_samples_leaf = 2)
mody3.fit(X, y3)

# Get the explainers and the SHAP values
```

```
## GradientBoostingRegressor(min_samples_leaf=2)
```

```
exy1 = shap.TreeExplainer(mody1)
y1ex = exy1.expected_value
shapy1 = exy1.shap_values(X)

exy2 = shap.TreeExplainer(mody2)
y2ex = exy2.expected_value
shapy2 = exy2.shap_values(X)

exy3 = shap.TreeExplainer(mody3)
y3ex = exy3.expected_value
shapy3 = exy3.shap_values(X)

preds3_real = mody1.predict(X) * mody2.predict(X)

preds3 = mody3.predict(X)
```

Computing the Multiplicative SHAP Values Using the Proposed Method

To aid in the computation of Multiplicative SHAP values, we have written a function in R which will return a list containing valuable information about the SHAP values. The code for the function is as follows:

```
multiply_shap <- function(shap1, shap2, ex1, ex2) {
  # Error Checking
  if (min(dim(shap1) == dim(shap2)) == FALSE) {
    stop("`shap1` and `shap2` must have the same dimensions")
  }
  if (length(ex1) > 1) {
    warning("`ex1` has a length greater than 1, only using first element")
    ex1 <- ex1[1]
  }
  if (length(ex2) > 1) {
    warning("`ex2` has a length greater than 1, only using first element")
    ex2 <- ex2[1]
  }

  d <- purrr::map_dfc(
    .x = 1:ncol(shap1),
    .f = ~{
      (shap1 %>% dplyr::pull(.x)) * c(ex2) +
        (shap2 %>% dplyr::pull(.x)) * c(ex1) +
        ((shap1 %>% dplyr::pull(.x)) * (shap2 %>% rowSums())) / 2 +
```

```
        ((shap1 %>% rowSums()) * (shap2 %>% dplyr::pull(.x))) / 2
    }
  ) %>%
    magrittr::set_colnames(stringr::str_c("s", 1:ncol(shap1)))

  expected_value <- d %>%
    rowSums() %>%
    `+`(ex1 * ex2) %>%
    mean()

  # return a list with what we want
  list(
    vals = d,
    ex1ex2 = ex1 * ex2,
    ex3 = expected_value,
    alpha = ex1 * ex2 - expected_value
  )
}
```

When applied to one of the objects from python, we get the following:

```
final_shap <- multiply_shap(
  shap1 = py$shapy1 %>% as.data.frame(),
  shap2 = py$shapy2 %>% as.data.frame(),
  ex1 = c(py$y1ex),
  ex2 = c(py$y2ex)
)
str(final_shap)
```

```
## List of 4
##  $ vals  : tibble [1,000 x 3] (S3: tbl_df/tbl/data.frame)
##   ..$ s1: num [1:1000] 153.6 -141.9 -162.3 -59.2 -68.3 ...
##   ..$ s2: num [1:1000] -190.8 176.3 161.9 133.1 23.2 ...
##   ..$ s3: num [1:1000] 34.18 -16.06 -33.72 -66.03 -4.08 ...
##  $ ex1ex2: num 58.5
##  $ ex3   : num 193
##  $ alpha : num -134
```

Distributing Alpha Introduction

This list object can then be passed to the following function that was written for distributing alpha, in any one of four different ways.

```
distribute_alpha <- function(multiplied_shap, method = "uniform") {
  if (method == "uniform") {
    d <- purrr::map_dfc(
      .x = multiplied_shap$vals,
      .f = ~{
        .x + (multiplied_shap$alpha / ncol(multiplied_shap$vals))
      }
    )
  } else if (method == "weighted_raw") {
    tot_s <- rowSums(multiplied_shap$vals)
    d <- purrr::map_dfc(
```

```
      .x = multiplied_shap$vals,
      .f = ~{
        .x + (.x / tot_s) * (multiplied_shap$alpha)
      }
    )
  } else if (method == "weighted_absolute") {
    tot_s <- rowSums(abs(multiplied_shap$vals))
    d <- purrr::map_dfc(
      .x = multiplied_shap$vals,
      .f = ~{
        .x + (abs(.x) / tot_s) * (multiplied_shap$alpha)
      }
    )
  } else if (method == "weighted_squared") {
    tot_s <- rowSums(multiplied_shap$vals^2)
    d <- purrr::map_dfc(
      .x = multiplied_shap$vals,
      .f = ~{
        .x + ((.x ^ 2) / tot_s) * (multiplied_shap$alpha)
      }
    )
  }

  d %>%
    dplyr::mutate(expected_value = multiplied_shap$ex3) %>%
    dplyr::mutate(predicted_val = rowSums(.))
}
```

Which when applied in the uniform case returns the following:

```
shap_unif <- final_shap %>% distribute_alpha("uniform")
head(shap_unif)
```

```
## # A tibble: 6 x 5
##       s1     s2     s3 expected_value predicted_val
##    <dbl>  <dbl>  <dbl>          <dbl>         <dbl>
## 1  109.  -235.   -10.5           193.        55.6
## 2 -187.   132.   -60.7           193.        76.9
## 3 -207.   117.   -78.4           193.        24.4
## 4 -104.    88.4 -111.            193.        66.4
## 5 -113.   -21.4  -48.7           193.         9.41
## 6  -31.1 -149.   -13.3           193.        -0.531
```

This dataframe contains the SHAP values for each variable, the expected value for the model output (based on the training data) and the predicted value (the sum of the expected value and all the SHAP values). With this, we have something that we can compare against what we might expect the SHAP values to be.

Methods for Distributing Alpha

Uniform Distribution

Weighted Distribution - Raw Value

Weighted Distribution - Squared Value

Weighted Distribution - Absolute Value

Assessment of Methods

```
def pred_fun(X):
  a = mody1.predict(X) * mody2.predict(X)
  return a

kernelex = shap.KernelExplainer(pred_fun, X)
```

```
## Using 1000 background data samples could cause slower run times. Consider using shap.sample(data, K)
```

```
real_shap_vals = kernelex.shap_values(X)
```

```
##   0%|          | 0/1000 [00:00<?, ?it/s]  0%|          | 1/1000 [00:00<03:34,  4.67it/s]  0%|
```

```
real_ex = kernelex.expected_value
```

Just to be noted, that took 4 minutes and 15 seconds, and it is an approximation. But now, we have what the kernel explainer would predict for the different SHAP values, we can import that into R and compare to what we obtained above.

We will also make use of a function to compute some basic information about the differences and similarities in each column

```
compare_shap_vals <- function(test_shap, real_shap) {
  purrr::map2_dfr(
    .x = test_shap %>% select(-expected_value, -predicted_val),
    .y = real_shap %>% select(-expected_value, -predicted_val),
    .f = ~{
      data.frame(
        mae = (sum(abs(.x - .y))) / length(.x),
        rmse = sqrt((sum((.x - .y)^2)) / length(.x)),
        pct_same_sign = mean((.x * .y) > 0)
      )
    }
  ) %>%
    mutate(
      variable = paste0("s", 1:nrow(.))
    )
}
```

Uniformly Distributed

```
real_shap <- py$real_shap_vals %>% as.data.frame() %>%
  magrittr::set_colnames(paste0("s", 1:ncol(.), "_real")) %>%
  mutate(expected_value = py$real_ex) %>%
  mutate(predicted_val = rowSums(.))
all_unif <- shap_unif %>%
```

```
  cbind(real_shap)

mean(abs(all_unif$predicted_val- all_unif$real_preds) < 0.000000001)
```

```
## [1] NaN
```

Weighted Distribution with the Raw Values

Weighted Distribution with the Squared Values

Weighted Distribution with the Absolute Values