Code review for WhyR??

Kacper Grzykowski, Dominik Kędzierski, Jakub Piwko

Kwiecień 2022

Wstęp

Niniejszy raport jest wynikiem code review funkcji budujących model XGBoost wykonanych przez grupę **WhyR??**. Kod osiąga cel, który postawiono. Przy użyciu funkcji napisanych przez naszych kolegów jesteśmy w stanie stworzyć model i wykorzystać go później do prognozowania interesujących nas informacji.

Analiza kodu

Na początek przyjrzeliśmy się bliżej samemu kodowi - jego stylowi, strukturze, logice oraz dokumentacji. Staraliśmy się znaleźć uwagi, które mogłyby polepszyć działanie kodu, lub ułatwić użytkownikowi korzystanie z funkcji.

Jeśli chodzi o styl i strukturę kodu, nie możemy niczego zarzucić. Kod jest schludny i przejrzysty. Bardzo łatwo się go czyta. Umieszczenie dodatkowych komentarzy przed każdym kolejnym krokiem preprocessingu czy budowania modelu jest ogromnym plusem. Świetnie, że preprocessing danych został zrealizowany w odrębnej funkcji.

```
data_DT <- data.table(data)
data_enc <- model.matrix(~.+0,data = data_DT[,-c(target),with=F])
data_matrix <- xgb.DMatrix(data = data_enc,label = labels)
```

Bardzo podoba nam się, że autorzy nie używali kilka razy z tych samych nazw, tylko faktycznie nazywali każdy krok w preprocessingu. Oczywiście to zużywa więcej pamięci, ale znacznie lepiej się czyta i debuguje.

Jeśli chodzi o aspekt techniczny kodu, znaleźliśmy kilka drobnych mankamentów, które mogą przyczyniać się do komplikacji. Niektóre opiszemy już przy okazji testowania.

```
  # setting to data table format (recommended)
  data_DT <- data.table(data)
```

Zrozumiałe jest użycie rekomendowanej data.table, ale gdzieś trzeba zaznaczyć jakie są zależności. Wystarczyłoby umieścić gdzieś w pliku `library(data.table)` lub korzysta z kwalifikowanych importów `data.table::data.table()`. W Roxygen zaznacza się importowane pakiety w następujący sposób `@import data.table` 1. Dla naszego use case nie ma to dużego znaczenia, ale gdybyście chcieli zrobić z tego pakiet, to można byłoby o tym pomyśleć.

Parametry zostały bardzo wyczerpująco opisane w dokumentacji funkcji. Definicje są jasne i zwięzłe, od razu można zrozumieć do czego potrzebujemy danego parametru. Ogromnym plusem jest dodawanie do niektórych parametrów wskazówek jak najlepiej je stroić i z jakimi innymi parametrami je dobierać. Zdecydowanie ułatwia to ich tuning. Jednak, patrząc od strony praktycznej, mamy kilka spostrzeżeń co do ich wykorzystania w funkcjach.

```
params <- list(booster = booster, eta = eta,
                 gamma = gamma, max_depth = max_depth, min_child_weight = min_child_weight,
                 subsample = subsample, colsample_bytree = colsample_bytree,
                 lambda = lambda, alpha = alpha)
```

Jako, że wszystkie parametry używane w funkcji budującej model zostały zdefiniowane jako domyślne i autorzy nie zdecydowali się wykorzystać elipsy, użytkownik nie jest w stanie zmienić lub dodać innych parametrów. Przykładowo, jeśli chcielibyśmy wykonać model dla binarnej klasyfikacji, a nie domyślnej regresji, musielibyśmy sami zmienić strukturę funkcji, dodając `objective = 'binary:logistic'` w liście parametrów.

```
xgb_model <- xgb.train(params = params, data = dtrain_matrix, nrounds = 79)
```

Parametr **nrounds** nie jest wcale parametryzowany, a odgrywa istotną rolę przy trenowaniu modeli drzewiastych. To jego tuning często pomaga zapobiec przetrenowaniu.

Dokumentacja, jak już wspomnieliśmy przy okazji parametrów, jest wyczerpująca. Chociaż może okazać się, że zabrakło w niej kilku informacji, np. przykładów użycia funkcji predykcyjnej. Trzeba jednak zauważyć, ze wszystkie elementy, które zostały przedstawione podczas prezentacji, zostały uwzględnione podczas budowania kodu. Jak to jednak ma się do praktyki?

Testowanie

Udało nam odtworzyć zamieszczone przykłady użycia funkcji `XGBoost_function`. Jednak napotkaliśmy problemu przy próbie wykorzystania funkcji `XGB_predict`.

```
#' Function which is presented below transform data (inplace)
#'automatically to make it available to build model.
#'Then function is building XGB model with default hyperparameters
#'(if neither is given) or specified hyperparameters if any are given,
#'fitting data to model and returning model. Then we can use
#'this model to predict the data we are interested in.
#' @param data
#'There we need to give data based on which we will predict.
#' It is well known as 'X' in machine learning literature.

XGB_predict <- function(data, model) {
  pred <- predict(model, data)
  return(pred)
}
```

Zarówno w dokumentacji, jak i przykładach nie ma informacji w jakim formacie przekazać dane. W samej funkcji także brak preprocessingu danych, a funkcja `XGB_predict` przyjmuje jedynie format macierzy rowych, rzadkich lub xgb.matrix. Musieliśmy pamiętać, aby przed przewidywaniem wartości odpowiednio dopasować dane testowe.

```
df2 <- data.frame(read.csv('breast-cancer.csv'))
```

Co do danych, na których konstruowane były przykłady, wygodniej byłoby przeprowadzić je na zbiorach, które można załadować prosto z R, ewentualnie z jakiegoś pakietu.

Przy próbie zbudowania modelu dla naszych danych, napotkaliśmy na pewien problem. Model nie chciał się zbudować

```
Błąd w poleceniu '`contrasts<-`(`*tmp*`, value = contr.funs[1 + isOF[nn]])':
 contrasts can be applied only to factors with 2 or more levels
```

Wynika to z tego, że dane wejściowe nie mogą zawierać wartości NA. Po ręcznym usunięciu wszystkich kolumn, które zawierają takie wartości, nasz model zbudował się poprawnie.

A przekazać dane do predykcji musieliśmy równie sami zrobić preprocessing tak, aby funkcja `XGB_predict` zadziałała prawidłowo.

Gdy chcemy wykonać predykcje nie ma informacji, w jakiej postaci mamy przekazać dane, natomiast próba przekazania ramki danych owocuje wystąpieniem błędu. Przekazanie danych testowych do funkcji preprocessingu także nie pomaga w tej kwestii.
