Badania eksploracyjne

Agata Rogowska, Zuzanna Różak, Magdalena Mazur, Aleksandra Siepiela

8 listopada 2020

Badanie eksploracyjne danych (ang. exploratory data analysis) dotyczy opisu, wizualizacji i badania zebranych danych bez potrzeby zakładania z góry hipotez badawczych. Badania ekploracyjne obejmują również wstępne sprawdzenie danych w celu skontrolowania założeń modeli statystycznych lub występowania błędów w danych (np. braków odpowiedzi).

Dane tabelaryczne

Dane tabelaryczne to dane, które mają postać tabeli. Tabela to struktura danych, która składa się z wierszy i kolumn. Każdy wiersz odpowiada pewnej obserwacji, której cechy zostały zapisane w kolejnych kolumnach.

Typy zmiennych

Zmienne, które opisują kolejne obserwacje możemy podzielić na:

- zmienne jakościowe (niemierzalne)
  - porządkowe - np. klasyfikacja wzrostu (niski, średni, wysoki)
  - nominalne - np. kolor oczu, płeć, grupa krwi
- zmienne ilościowe (mierzalne)
  - dyskretne - np. liczba dzieci, liczba gospodarstw domowych, wiek (w rozumieniu ilości skończonych lat)
  - ciągłe - np. wzrost, masa, wiek (w rozumieniu ilości dni między datą urodzin a datą badania)
  - proporcjonalne - np. masa, długość, temperatura wyrażona w Kelwinach lub stopniach Rankine'a (przyjmujemy istnienie zera i możemy twierdzić, że jedno ciało jest dwukrotnie gorętsze od drugiego)
  - interwałowe - np. temperatura wyrażona w stopniach Celsjusza lub Fahrenheita (możemy twierdzić, że coś jest o 20 °C cieplejsze od czegoś innego, ale nie możemy stwierdzić ilokrotnie cieplejsze jest ciało o temperaturze 40 °C od ciała o temperaturze –10 °C), data kalendarzowa (możemy mówić o stałej różnicy pomiędzy kolejnymi dniami)

Miary

Zapoznając się z danymi chcielibyśmy sprawdzić wokół jakiej wartości są skupione oraz jak bardzo są zmienne wartości danej cechy.

Miary lokacji (miary tendencji centralnej) pomagają nam umiejscowić dane na osi. Przykładami takich miar są:

- średnia - najczęściej arytmetyczna określona jako $\overline{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$.
- dominanta (moda) - ozn. Mo - dla zmiennych o rozkładzie dyskretnym, wartość o największym prawdopodobieństwie wystąpienia lub wartość najczęściej występująca w próbie. Dla zmiennej losowej o rozkładzie ciągłym jest to argument, dla którego funkcja gęstości prawdopodobieństwa ma wartość największą.

- mediana - ozn. Me - wartość cechy w szeregu uporządkowanym, powyżej i poniżej której znajduje się jednakowa liczba obserwacji.
- kwantyle rzędu p - wartość cechy w szeregu uporządkowanym, poniżej której znajduje się $p \cdot 100\%$ liczby obserwacji, a powyżej której znajduje się $(1-p) \cdot 100\%$ liczby obserwacji.

Natomiast miary rozrzutu dostarczają informacji jak bardzo zróżnicowane są obserwacje pod względem badanej cechy. Przykładami takich miar są:

- wariancja - stopień rozrzutu badanej cechy wokół wartości oczekiwanej. Im większa wariancja, tym rozrzut zmiennej jest większy. Nieobciążony estymator wariancji wyraża się wzorem: $s^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \overline{x})^2$
- odchylenie standardowe - mówi nam o przeciętnym odchyleniu wartości zmiennej losowej od jej wartości oczekiwanej. Im odchylenie standardowe jest większe, tym większe zróżnicowanie wartości badanej cechy. Odchylenie standardowe z próby obliczamy jako pierwiastek z wariancji z próby, tzn. $s = \sqrt{s^2}$.
- rozstęp międzykwartylowy - różnica między trzecim a pierwszym kwartylem. Ponieważ pomiędzy tymi kwartylami znajduje się z definicji 50% wszystkich obserwacji (położonych centralnie w rozkładzie), dlatego im większa szerokość tego rozstępu, tym większe zróżnicowanie cechy.

Wyróżniamy także miary asymetrii. Miary asymetrii mówią nam, czy większa część populacji klasuje się powyżej, czy poniżej przeciętnego poziomu badanej cechy. Asymetrię rozkładu można zbadać porównując średnią, dominantę i medianę.

- W przypadku rozkładu symetrycznego wszystkie te parametry są równe.
- Jeśli zachodzi nierówność $Mo < Me < \mathbb{E}X$, to mamy do czynienia z prawostronną asymetrycznością rozkładu. Tzn. dużo małych wartości i bardzo mało dużych.
- Jeśli zachodzi nierówność $\mathbb{E}X < Me < Mo$, to mamy do czynienia z lewostronną asymetrycznością rozkładu. Tzn. mało małych i bardzo dużo dużych.

R - podsumowanie kolumn

Podstawowymi funkcjami, które pomagają nam zapoznać się z danymi są funkcje:

- `head` - zwraca pierwszą część wektora, macierzy, tabeli lub ramki danych. Domyślnie 6 pierwszych elementów.
- `nrow` - zwraca liczbę wierszy macierzy, tabeli lub ramki danych.
- `ncol` - zwraca liczbę kolumn macierzy, tabeli lub ramki danych.

Natomiast podstawowymi funkcjami, które podsumowują kolejne kolumny są funkcje:

- `str` - zwraca strukturę danego obiektu. Wyświetla np. klasę obiektu, liczbę wierszy i kolumn, a także nazwę danej kolumny, typ wartości w niej zawartych, jak i kilka początkowych wartości.
- `summary` - zwraca podsumowanie każdej kolumny. Dla zmiennych ciagłych wyznacza wartości tj.:
  - wartość najmniejsza i największa
  - średnia i mediana
  - 1 (0.25) i 3 (0.75) kwartyl
  - liczba wartości brakujących (NA)

Natomiast w przypadku zmiennych dyskretnych wyznacza liczbę obserwacji, które przyjmują daną wartość zmiennej.

- `glimpse` - funkcja z pakietu `tidyverse` podobna do `str`, ale stara się pokazać jak najwięcej danych. Wyświetla np. liczbę wierszy i kolumn, a także nazwę danej kolumny, typ wartości w niej zawartych oraz jak najwięcej wartości z tej kolumny.

R - funkcje

Funkcje przydają się do zamknięcia w nich operacji, które się często powtarzają w naszym kodzie lub dla jego lepszej czytelności. Podstawowa składnia funkcji w R wygląda tak:

```
nazwa_funkcja <-  function(argument 1, argument 2, ...){
  ciało funkcji
  return(wartość lub obiekt zwracany)
}
```

Napiszmy funkcję, która będzie mnożyła dowolny wektor przez podaną liczbę, a następnie zsumuje elementy wektora:

```
funkcja1 <- function(wektor, liczba){
  rezultat <- wektor * liczba
  rezultat <- sum(rezultat)
  return(rezultat)
}
```

Możemy także pominąc `return` i zdefiniować funkcje:

```
funkcja2 <- function(wektor, liczba){
  rezultat <- wektor * liczba
  rezultat <- sum(rezultat)
  rezultat
}
```

Obie funkcje `funkcja1` i `funkcja2` robią to samo. Wykonajmy nasze funkcje dla dwóch zdefiniowanych zmiennych:

```
v <- 1:5
n <- 2

funkcja1(v, n)
```

```
## [1] 30
```

```
funkcja2(v, n)
```

```
## [1] 30
```

Oczywiście do wykonania funkcji potrzebne jest zdefiniowanie obu argumentów. Jak ich nie dodamy wyświetli się błąd, że argument drugi zaginął i nie mamy zdefiniowanej jego wartości domyślnej. Zdefiniujmy zatem domyślną wartość argumentu `liczba` jako `NULL` i dopiszmy do naszej funkcji kod, który gdy ten argument będzie miał wartość domyślną zwróci tylko sumę elementów wektora:

```
funkcja3 <- function(wektor, liczba = NULL){
  if(is.null(liczba)){
    rezultat <- sum(wektor)
  } else{
    rezultat <- wektor * liczba
    rezultat <- sum(rezultat)
  }
  rezultat
}
```

Wykonajmy funckję `funkcja3` na wcześniej zdefiniowanym wektorze `v`:

```
funkcja3(v)
```

```
## [1] 15
```

Oprócz zdefiniowania wartości domyślnej argumentu poprzez trzy kropki możemy również dopuścić parametry dodatkowe. Zdefiniujmy funkcję z parametrami dodatkowymi:

```
funkcja4 <- function(wektor, liczba = NULL, ...){
  if(is.null(liczba)){
    rezultat <- sum(wektor, ...)
  } else{
    rezultat <- wektor * liczba
    rezultat <- sum(rezultat, ...)
  }
  rezultat
}
```

Wykonajmy funckję `funkcja4` usuwając wartości brakujące z nowo zdefiniowanego wektora:

```
v <- c(NA, 1, NA, 2:4, NA, 5)

v
```

```
## [1] NA  1 NA  2  3  4 NA  5
```

```
funkcja4(v, na.rm = TRUE)
```

```
## [1] 15
```

Funkcje są bardzo przydatne, gdy mamy do napisania długi skrypt. Pozwalają na podzielenie głównej części kodu na mniejsze kawałeczki, które kolejnemu użytkownikowi skryptu lub nam będzie łatwiej modyfikować.

Podsumowanie kursu z Data Camp

1. Podstawowe informacje

System kontroli wersji to narzędzie, które zarządza zmianami wprowadzanymi w plikach i katalogach w projekcie. Istnieje wiele systemów kontroli wersji. Przykładem takiego systemu jest Git. Jego mocne strony to:

- Nic, co jest zapisane w Git, nigdy nie jest tracone, więc zawsze możesz wrócić, aby zobaczyć, które wyniki zostały wygenerowane przez które wersje twoich programów.
- Git automatycznie powiadamia Cię, gdy Twoja praca koliduje z pracą innej osoby, więc jest trudniej (choć nie jest to niemożliwe) o przypadkowe nadpisanie pracy.
- Git może synchronizować pracę wykonywaną przez różne osoby na różnych komputerach.

Kontrola wersji nie dotyczy tylko oprogramowania: książki, artykuły, zestawy parametrów i wszystko, co zmienia się w czasie lub wymaga udostępnienia, może i powinno być przechowywane i udostępniane za pomocą czegoś takiego jak Git.

Każdy z projektów Git składa się z dwóch części: plików i katalogów, które tworzysz i edytujesz bezpośrednio, oraz dodatkowych informacji, które Git rejestruje o historii projektu. Połączenie tych dwóch rzeczy nazywa się repozytorium.

Git przechowuje wszystkie dodatkowe informacje w katalogu o nazwie `.git` znajdującym się w katalogu głównym repozytorium.

Podstawowe komendy

Używając Gita zapewne często będziemy chcieli sprawdzić stan swojego repozytorium. Aby to zrobić, użyjemy polecenie `git status`.

- `git status` - wyświetla listę plików, które zostały zmodyfikowane od czasu ostatniego zapisania zmian

Git ma obszar przejściowy, w którym przechowuje pliki ze zmianami, które chcemy zapisać, a które nie zostały jeszcze zapisane.

- `git status` - pokazuje, które pliki znajdują się w tym obszarze przejściowy i które mają zmiany, które nie zostały jeszcze zatwierdzone
- `git diff` - pokaże wszystkie zmiany w twoim repozytorium (porównując obecną postać plików z ostatnio zapisaną)
- `git diff directory` - pokaże zmiany w plikach w jakimś katalogu (porównując obecną postać plików z ostatnio zapisaną)
- `git diff filename` - pokaże zmiany w danym pliku (porównując obecną postać z ostatnio zapisaną)

Git różnice między dwiema wersjami pliku wyświetla w poniższy sposób:

```
diff --git a/report.txt b/report.txt
index e713b17..4c0742a 100644
--- a/report.txt
+++ b/report.txt
@@ -1,4 +1,5 @@
-# Seasonal Dental Surgeries 2017-18
+# Seasonal Dental Surgeries (2017) 2017-18
+# TODO: write new summary
```

gdzie:

- `a/report.txt, b/report.txt` to pierwsza i druga wersja pliku,
- linia druga wypisuje klucze do wewnętrznej bazy danych zmian Gita,
- `-- a/report.txt, +++ b/report.txt` oznacza, że usuwane linie oznaczone są przedrostkiem `-`, dodawane linie oznaczone są przedrostkiem `+`,
- linia zaczynająca się od `@@` mówi, gdzie wprowadzane są zmiany. Pary liczb to numer lini „startowej" i liczba linii,
- kolejne linie są listą zmian, które zostały wprowadzone.
- `git add filename` - dodaje plik do obszaru przejściowego
- `git diff -r HEAD` - porówna pliki z repozytorium z plikami z obszaru przejściowego
- `git diff -r HEAD path/to/file` - porówna konkretny plik z repozytorium z plikiem z obszaru przejściowego
- `nano filename` - otwiera plik w edytorze tekstowym `nano`
  - poruszanie się strzałkami
  - `Backspace` - usuń znak
  - `Ctrl-K`: usuń linię
  - `Ctrl-U`: cofnij usunięcie linii
  - `Ctrl-O`: zapisz plik
  - `Ctrl-X`: wyjdź z edytora
- `git commit -m "comment"` - zapisuje zmiany w obszarze przejściowym z jednowierszowym komunikatem o wprowadzonych zmianach

- `git commit -amend - m "new message"` - zmienia ostatni komunikat
- `git log` - wyświetlenie historii projektu (od najnowszych zmian). Wyświetlany zostaje unikatowy identyfikator dla zatwierdzenia oraz informacje na temat tego kto dokonał zmiany, kiedy i jaki komunikat napisał dokonując zmiany.
  - `spacja` - przejcie w dół o stronę
  - `q` - wyjście
- `git log path` - wyświetlenie historii danego pliku lub katalogu

2. Repozytoria

Informacje dotyczące zatwiedzonych zmian przechowywane są poprzez trzypoziomową strukturę. Każde zatwierdzenie (tzw. *commit*) zwiera komunikat o zatwierdzeniu i informacje o autorze i czasie, w którym zatwierdzenie zmian zostało wykonane. Każdy *commit* ma również swoje drzewo, które śledzi, gdzie w repozytorium dokonano zmian. Dla każdego pliku w drzewie istnieje tzw. *blob* (*binary large object*). Każdy *blob* zawiera skompresowaną migawkę zawartości pliku, z chwili w której nastąpił *commit*.

Czym jest hash?

Każde zatwierdzenie zmian w repozytorium ma unikalny identyfikator zwany **hash'em**. Jest on zapisywany jako 40-znakowy ciąg szesnastkowy. Zazwyczaj jednak wystarczy podać pierwsze 6 lub 8 znaków hash'a, by odnaleźć konkretne zatwierdzenie (*commit*). Identyfikatory jakimi są hash'e umożliwiają Git'owi wydajne udostępnianie danych pomiędzy repozytoriami.

Jak wyświetlić konkretny commit?

By wyświetlić szczegóły dotyczące konkretnego *commit'u* należy użyć komendy **git show** z pierwszymi 6 znakami *hash'a* danego *commmit'u* np.: *git show Oda2f7*.

Czym jest odpowiednik ścieżki względnej w Git?

Innym sposobem identyfikacji zatwierdzenia jest użycie odpowiednika ściezki względnej. By wyświetlić zatem ostatni *commit* możemy użyć komendy *git show* z etykietą *HEAD*. Jeśli natomiast zamiast *HEAD* wpiszemy *HEAD~1* wyświetlony zostanie przedostatni *commit*, polecenie *git show HEAD~2* zwróci nam natomiast jeszcze wcześniejszy *commit* itp.

Podstawowe komendy

- **git log** - wyświetla całą historię danego pliku lub projektu. W Gic'ie możemy jednak sprawdzić bardziej szczegółowe informacje. Dzięki poleceniu **git annotate file** możemy sprawdzić kto i kiedy dokonał ostatniej zmiany w każdej linijce pliku.
- **git diff ID1..ID2** - umożliwia sprawdzenie zmian pomiędzy dwoma commitami, których identyfikatory to odpowiednio ID1 i ID2.
- **git add** - polecenie umożliwiające dodanie nowego pliku. Po wykonaniu tego polecenia Git zaczyna śledzić dodany plik.
- **git clean -n** - pokazuje listę plików, które są w repozytorium, ale których historia nie jest śledzona przez Git'a.
- **git clean -f** - usuwa pliki, które są w repozytorium i których historii nie śledzi Git. Z używaniem tego polecenia należy uważać, ponieważ usuwa ono pliki z pamięci na stałe i nie da się ich już odzyskać.
- **git config - -list** - wyświetla ustawienia Git'a.
- **git config - -system** - wyświetla ustawienia każdego użytkownika na danym komputerze.
- **git config - -global** - wyświetla ustawienia każdego projektu.

- **git config - -local** - wyświetla ustawienia poszczególnego projektu.

Każdy poziom zastępuje poziom nad nim, więc ustawienia lokalne (na projekt) mają pierwszeństwo przed ustawieniami globalnymi (na użytkownika), które z kolei mają pierwszeństwo przed ustawieniami systemowymi (dla wszystkich użytkowników na komputerze).

- **git config - -global setting value** - zmienia konfigurację odpowiedniej wartości dla wszystkich projektów na danym komputerze. Jako *setting* należy wpisać to co chcemy zmienić (np. *user.name*, *user.email* itp.), a jako *value* to co chcemy ustawić.

3. Cofanie zmian

Teraz dowiemy się jak cofnąć wprowadzone zmiany.

- `git reset HEAD`- usuwa ostatnio dodany plik ze śledzenia,
- `git checkout - filename` - odrzuci zmiany, które nie zostały jeszcze dodane do śledzenia,
- `git reset HEAD path/to/file` - odrzuci ostatnie zmiany w pliku, który został juz dodany do śledzenia,
- `git checkout 2242bd filename`- zamienia aktualna wersje pliku, na tę o hashu '2242bd'.

Do ostatniej komendy przydatne może być wykonanie poniższzego polecenia, aby sprawdzić hashe plików.

- `git log - 3 filename`- pokaże 3 ostatnie commity dotyczące wskazanego pliku.

Poniższe dwie komendy pokazują, jak cofać zmiany na więcej niż jednym pliku.

- `git reset HEAD data`- usuwa ze śledzenia wszystkie pliki z katalogu data. Jeżeli nie podamy nazwy katalogu(wtedy wystarczy samo `git reset`) wszystkie pliki zostaną usunięte.
- `git checkout - data`- wszystkie pliki w katalagu data zostaną cofnięte do poprzednich wersji.

4. Gałęzie

Jeśli nie używasz kontroli wersji, typowym przepływem pracy jest tworzenie różnych podkatalogów do przechowywania różnych wersji projektu w różnych stanach, na przykład deweloperskich i końcowych. Oczywiście zawsze kończy się to ostateczną aktualizacją i ostateczną aktualizacją-poprawioną. Problem polega na tym, że trudno jest to rozwiązać, jeśli masz odpowiednią wersję każdego pliku w odpowiednim podkatalogu i ryzykujesz utratę pracy.

Jednym z powodów, dla których Git jest popularny, jest jego obsługa tworzenia gałęzi (*branchy*), co pozwala na posiadanie wielu wersji Twojej pracy i pozwala na systematyczne śledzenie każdej wersji.

Każda gałąź jest jak wszechświat równoległy: zmiany, które wprowadzasz w jednej gałęzi, nie wpływają na inne gałęzie (dopóki nie połączysz ich z powrotem).

Domyślnie kazde repozytorium Gita ma branch zwany **master**.

Podstawowe komendy związanie z działaniem na branchach (gałęziach):

- `git branch` - pokazuje wszystkie branche w repozytorium (branch, w którym obecnie się znajdujesz będziesz wylistowany z $*$).
- `git diff branch1..branch2` - wyświetla różnice między dwoma branchami

Ciekawostka:

- `git diff branch1..branch2 - -shortstat` - wyświetla konkretną liczbę plików które się różnią między dwoma branchami

- `git checkout branch1` - pozwala przełączyć się na *branch1*
- `git checkout -b branch-name` - pozwala utworzyć nowego brancha o nazwie *branch-name*

  Rozgałęzianie pozwala tworzyć równoległe wszechświaty. Scalanie (**merging**) to sposób, w jaki łączysz je z powrotem. Kiedy łączysz jedną gałąź (nazwijmy ją źródłową) z inną (nazwijmy ją docelową), Git włącza zmiany wprowadzone w gałęzi źródłowej do gałęzi docelowej. Jeśli te zmiany nie nakładają się, wynikiem jest nowe zatwierdzenie w gałęzi docelowej, które zawiera wszystko z gałęzi źródłowej. Do mergowania dwóch gałęzi używamy polecenia:

- `git merge source destination` - mergowanie dwóch branchy w jeden

Czasami zmiany w dwóch gałęziach będą ze sobą kolidować: na przykład poprawki błędów mogą dotyczyć tych samych wierszy kodu lub analizy w dwóch różnych gałęziach mogą dołączać nowe (i różne) rekordy do pliku danych podsumowania. W takim przypadku ty decydujesz o sprzeczności zmian.

Jeżeli podczas mergowania występuje konflikt Git informuje Cię, że wystapił problem a `git status` poinformuje Cię, które pliki wmagają rozwiązania konfliktów.

Git pozostawia na danym pliku znaczniki, aby poinformować Cię o konkretnym miejscu konfliktu. Znaczniki te wyglądają następująco:

```
<<<<<<< destination-branch-name
...changes from the destination branch...
=======
...changes from the source branch...
>>>>>>> source-branch-name
```

Aby rozwiązać konflikt edytuj plik, usuwając znaczniki i wprowadź wszelkie zmiany potrzbne do rozwiązania kofilktu, a następnie zrób *commit* tych zmian.

5. Tworzenie własnych repozytoriów

Przejdźmy do kolejnego zagadnienia związanego z pracą w Gicie.

Do tej pory wszystkie poznane funkcje Gita dotyczyły działań na repozytoriach już istniejących. Aby stworzyć własne repozytorium w bieżącym katalogu roboczym wystarczy komenda:

- `git init project-name`

Warto wspomnieć, że chociaż Git pozwala tworzyć zagnieżdżone repozytoria nie powinieneś tego robić. Aktualizacja takich repozytoriów bardzo szybko staje się bardzo skomplikowana, ponieważ musisz powiedzieć Gitowi, w którym z dwóch katalogów .git ma być przechowywana aktualizacja.

Nie tworzymy repozytorium w innym już istniejącym!

Poniżej kilka ważnych komend:

- `git init` - inicjalizacja repozytorium w bieżącym katalogu
- `git init /path/to/project` - inicjalizacja repozytorium we wskazanym ścieżką katalogu
- `git clone URL` - tworzenie kopii istniejącego pod wskazanym adresem *URL* repozytorium
- `git clone /existing/project newprojectname` - tworzenie kopii istniejącego repozytroium o zadanej nazwie - *newprojectname*

- `git remote` - wyświetla informację o fizycznej lokalizacji na serwerze Gita, z której zostało sklonowane repo
- `git remote -v` - wyświetla informację o *URL* serwerze Gita, z którego zostało sklonowane repo
- `git remote add remote-name URL` - pozawala na dodanie własnego remota z podanego *URL*
- `git remote rm remote-name` - usuwanie istniejącego remota
- `git pull remote branch` - pobieranie zmian w *branchu* w lokalnym repozytorium i mergowanie ich z bieżacym brnachem w lokalnym repozytorium

Uwaga!

Git powstrzymuje Cię przed pobieraniem ze zdalnego repozytorium zmian, które mogą nadpisać niezapisane lokalnie zmiany. Wystarczy zrobić commit *tych zmian lub cofnąć je, a następnie spullować repo ponownie.*

- `git push remote-name branch-name` - pushuje zmiany wprowadzone lokalnie na danym branchu do zdalnego repozytorium