Wprowadzenie do wykładu

Cele bieżącego wykładu

- Wprowadzenie do zagadnień implementacji algorytmów sztucznej inteligencji
- Ogólne omówienie języków programowania najczęściej wykorzystywanych do programowania sztucznej inteligencji (np. Python, Prolog, R, Julia)
- Konfiguracja środowiska programistycznego i podstawowych narzędzi, w tym do zarządzania konfiguracją

Czego potrzeba, żeby programować sztuczną inteligencję?

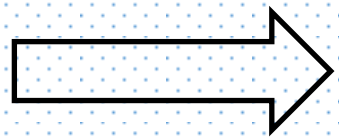
- Choć na rynku pracy istnieje spore zapotrzebowanie na osoby z umiejętnościami programowania SI, to wcale nie znaczy, że każdy od razu znajdzie (dobrą) pracę w tej dziedzinie!
- Nauka ciągle idzie do przodu, wymyślane są nowe algorytmy i techniki, a stare są udoskonalane
- Technologia też wciąż się rozwija, powstają nowe narzędzia, biblioteki oraz techniki programowania
- Mamy coraz więcej danych do przetworzenia

Trzeba zatem systematycznego i uporządkowanego podejścia do zagadnienia, a przede wszystkim odpowiednich

umiejętności i pasji!

Kluczowe umiejętności (1)

- **Solidne fundamenty**
  - Opanowanie podstaw programowania w wybranym języku programowania (omówienie najpopularniejszych z nich znajdzie się w dalszej części prezentacji)
  - Opanowanie podstaw matematyki wyższej (algebra liniowa, statystyka, rachunek różniczkowy, rachunek prawdopodobieństwa)
  - Abstrakcyjne myślenie



- Python
- R
- Prolog
- Julia
- LISP
- Scala

Kluczowe umiejętności (2)

- **Rozumienie aparatu matematycznego**
  - W zagadnieniach związanych z programowaniem sztucznej inteligencji nie zawsze możliwe jest wykorzystanie funkcji bibliotecznych (co zwykle zwalniałoby z konieczności samodzielnego implementowania obliczeń matematycznych)
  - Programiści sztucznej inteligencji muszą zatem rozumieć złożone algorytmy obliczeniowe, żeby być gotowymi na ich samodzielne implementowanie w wybranym języku programowania
  - W szczególności istotne jest dobre rozumienie statystyki i rachunku prawdopodobieństwa, kluczowych w wielu zagadnieniach uczenia maszynowego

Kluczowe umiejętności (3)

- **Myślenie abstrakcyjne**
  - Polega (m. in.) na umiejętnościach zrozumienia ukrytych relacji między rzeczami
  - Rozumienie problemów oparte na takich relacjach pozwala na lepsze zrozumienie niuansów i złożoności metod stosowanych w sztucznej inteligencji, a w konsekwencji na lepsze, bardziej elastyczne ich programowanie
  - Jest dużo bardziej „rozmyte” niż samo uczenie się statystyki, matematyki czy logiki, ale nie mniej ważne

Kluczowe umiejętności (4)

- **Intuicja**
  - Intuicja pozwala na szybszą i bardziej efektywną pracę przy programowaniu SI
  - Można ją nabyć poprzez samodzielną realizację (implementację) prostych algorytmów od podstaw
  - Może to nie być łatwe na początku, ale da długoterminowe korzyści
  - Przykładowo, implementacja perceptronu prostego pozwoli na zrozumienie, na czym polega proces jego uczenia, co można później rozszerzyć na sieć neuronową
  - Implementacja systemu wnioskowania bayesowskiego pozwoli na lepsze zrozumienie praktycznych aspektów rachunku prawdopodobieństwa

Kluczowe umiejętności (5)

- **Umiejętność pozyskiwania danych**
  - Sztuczna inteligencja doskonale radzi sobie z przetwarzaniem dużych ilości danych jednocześnie
  - Już na etapie planowania systemu SI, warto przewidzieć zadania, dzięki którym będą pobierane dane ze wszelkich możliwych źródeł (m.in. nie tylko technologicznych ale też takich jak np. obsługa klienta i marketing)
  - Lepiej pozyskać więcej danych, niż za mało
  - Uwaga na etyczne zagadnienia związane ze sztuczną inteligencją i przetwarzaniem danych!

I właśnie nabywanie tych umiejętności będzie jednym z ważniejszych aspektów wykładu i całego przedmiotu

Python – wstęp

- Zaprojektowany w 1991 roku jako język ogólnego przeznaczenia, pierwsza stabilna wersja została wydana w 1994 r., wersja (gałąź) 2 w r. 2000 a wersja (gałąź) 3 – w 2008; darmowy (Open Source / GNU GPL)
- Interpretowany, obiektowy
- Stopniowo podbił społeczność naukową i wyrósł na dojrzały ekosystem specjalistycznych pakietów do przetwarzania i analizy danych
- Pozwala na eksperymenty i szybką, łatwą implementację teorii i szybkie wdrożenia aplikacji
- Inne zastosowania: strony WWW (framework Django, CMS Plone), systemy automatyki domowej (Home Assistant), mikrokontrolery (MicroPython), skrypty do administracji systemami komputerowymi i wiele innych

Python – wydajność

- Jest bardzo uniwersalny. Można programować w różnych stylach (obiektowo lub proceduralnie), niezależnie od poziomu umiejętności
- Jest wieloplatformowy – działa płynnie na systemach operacyjnych Windows, Linux i Mac
- Chociaż interpretowany, jest stosunkowo szybki (choć oczywiście sporo wolniejszy od C czy Javy)
- Są też szybkie implementacje, porównywalne z językami kompilowanymi – np. darmowy Intel Python, PyPy
- Można też pisać w języku Cython, który jest nadzbiorem Pythona i zapewnia wydajność języka C

Python – zalety

- Może pracować na dużych danych umieszczonych w pamięci, ze względu na jej minimalne zużycie i doskonałe nią zarządzanie: posiada efektywny odśmiecacz
- Jest dość prosty do nauczenia i używania
- Bez problemu przetwarza bardzo duże liczby oraz liczby w systemie dwójkowym, ósemkowym, szesnastkowym
- Łatwo przetwarza nowoczesne języki formalne do reprezentowania danych (np. YAML)
- Ma bogatą bibliotekę standardową i wiele dodatkowych modułów, z różnych dziedzin nauki i techniki
- Istnieją specjalizowane dystrybucje (czyli sam język + wygodny edytor + gotowe do wykorzystania biblioteki) do analizy i przetwarzania danych, np. Anaconda
- Istnieją specjalizowane „powłoki” (ang. *shells*), np. powłoka IPython, udostępniająca interfejs w postaci tzw. notebooka wyświetlanego w przeglądarce WWW

Python – popularne biblioteki wspierające programowanie SI

W nawiasie motto, określające jednym zdaniem główne zastosowanie

- SciPy
  - NumPy (ang. *Base N-dimensional array package* – podstawowy pakiet do obsługi N-wymiarowych tablic)
  - pandas (ang. *Data structures & analysis* – struktury i analiza danych)
  - Matplotlib (ang. *Comprehensive 2-D plotting* – kompleksowa obsługa wykresów 2-D)
- scikit-learn (ang. *Machine Learning in Python* – uczenie maszynowe w Pythonie)
- TensorFlow (ang. *An end-to-end open source machine learning platform* – kompleksowa platforma uczenia maszynowego typu open source)
- Keras (ang. *Deep learning API* – API do uczenia głębokiego)

Python – wady

- Dwie gałęzie, 2.x, 3.x, niekompatybilne ze sobą (jeżeli ktoś dopiero zaczyna pracę z Pythonem, to zdecydowanie zalecane jest wykorzystywanie gałęzi 3.x – gałąź 2.x nie jest już wspierana)
- Oznaczanie bloków programu za pomocą wcięć może być uciążliwe przy większych programach
- Niekompatybilność tabulatorów i spacji przy wcięciach
- Czasem niespójna składnia
- Częsta niekompatybilność zewnętrznych modułów między sobą (typowy problem: moduł *X* wymaga modułu *Z* w wersji co najmniej z_1, a moduł *Y* wymaga modułu *Z* w wersji co najwyżej z_2. Co, jeżeli $z_1 > z_2$?)

Python – przykład kodu

```
import os
import matplotlib.pyplot as plt

PERSONS = 30
TARGET_DIR = "ica_clustered_10"

# Collect directories
dirlist = []
for dir in os.listdir(TARGET_DIR):
    if os.path.isdir(TARGET_DIR + os.path.sep + dir):
        dirlist.append(dir)
dirlist.sort()

dist = [0 for i in range(PERSONS+1)]
```

R

- Szczególnie popularny wśród matematyków / statystyków
- Bardzo bogata biblioteka naukowa
- Łatwe programowanie symboli i wzorów matematycznych
- Interpretowany
- Świetnie nadaje się do wizualizacji wyników
- Trudniejszy do nauczenia od Pythona i nie jest językiem ogólnego zastosowania
- Darmowy (GNU GPL), pierwsza wersja w 2000 r.

R – przykład kodu

```
select_neighbors_id <- function(variables = NULL, distance =
gower::gow_dist, n = 50, frac = NULL) {
  if (is.null(variables)) {
    variables <- intersect(colnames(observation),
                           colnames(data))
  }
  if (is.null(n)) {
    n <- ceiling(nrow(data)*frac)
  }

  distances <- distance(observation[,variables, drop = FALSE],
                        data[,variables, drop = FALSE])
  head(order(distances), n)
}
```

Prolog

- Język deklaratywny, opisujący sposób sterowania algorytmem wnioskowania
- Wnioskowanie polega na zastępowaniu stwierdzeń, których prawdziwość jest poddawana weryfikacji przez inne stwierdzenia, zgodnie z zapisanymi definicjami tzw. predykatów
- Prolog realizuje tzw. wnioskowanie wstecz
- Wnioskowanie to kończy się, gdy algorytm wnioskowania dojdzie do stwierdzenia zapisanego jako prawdziwe, bądź wtedy, gdy nie da się już dopasować żadnej reguły wnioskowania
- Stary, pierwsze implementacje w 1970 r.

Prolog – przykład kodu

```
select(H,[H|T],T).
select(X,[H|T],[H|T1]) :- select(X,T,T1).

miejsce(W,D,A,B,G):-
ListaMiejsc=[1,2,3,4,5], /* Początkowa lista miejsc */
G = 3, /* Grzegorz przybiegł trzeci */
select(W,ListaMiejsc,L1), W\=1, /* Wincenty nie był pierwszy */
select(D,L1,L2), D\=2, /* Dymitr nie był drugi */
select(A,L2,L3), A\=1,A\=5, /* Andrzej nie był pierwszy ani
ostatni, skreślamy go */
select(B,L3,[G]), /* Zostali tylko Borys i Grzegorz */
B is W+1, /* Borys Przybiegł po Wincentym */
G>D. /* Dymitr przybiegł przed Grzegorzem */

/* miejsce(W,D,A,B,G). */
```

Julia

- Stosunkowo nowy (2012 r.)
- Wysokowydajny, dynamiczny, ogólnego przeznaczenia
- Podobnie jak R – łatwo się w nim programuje wzory matematyczne
- Bardzo łatwo łączy się go z Pythonem, C/C++, R, Fortranem, Javą…
- Wydajność zbliżona do języka C
- Bardzo łatwo poddać kod zrównolegleniu
- Bardzo popularny wśród młodych pracowników nauki

Julia – przykład kodu

```
function Basics.integrateOnGridNewtonCotes(F::Array{Float64,1}, grid::Radial.Grid)
    coefficients = Array{Float64}([2 * 7, 32, 12, 32, 7]) * 2 / 45 * grid.h
    n = size(coefficients, 1)
    result = 0.;    i0     = 1
    while  abs(F[i0]) == 0
       i0 += 1
    end
    gamma  = log(F[i0]/F[i0+1] * grid.rp[i0+1]/grid.rp[i0]) / log(grid.r[i0]/grid.r[i0+1])
    result = 1/(gamma + 1) * grid.r[i0] * F[i0] / grid.rp[i0]
    return( result )
  end
```

Lisp

- Jeden z najstarszych języków programowania (1958 r.) – starszy jest tylko Fortran
- Geneza nazwy: LISt Processing
- Współczesne dialekty: Common Lisp, Scheme, Clojure
- Posiada wygodną matematyczną notację dla programów komputerowych, opartą na tzw. rachunku lambda
- Kod tworzony jest jako struktura danych (w postaci tzw. S-wyrażeń)
- Od samego początku swojego istnienia był szeroko stosowany przez społeczność badającą i rozwijającą sztuczną inteligencję
- Traktowany od początku jako bardzo innowacyjny

Lisp – przykład kodu

```
(defun fibonacci (N)
 "Compute the N'th Fibonacci number."
 (if (or (zerop N) (= N 1))
 1
 (let
 ((F1 (fibonacci (- N 1)))
 (F2 (fibonacci (- N 2))))
 (+ F1 F2))))

(let
     ((x 1)
      (y (* x 2)))
  (+ x y))
```

Scala

- Stosunkowo młody (2001 r.)
- Nazwa podkreśla skalowalność języka (ang. *scalable language*)
- Wspiera wielowątkowość, pozwalając na tworzenie wysokowydajnych aplikacji klasy Enterprise
- Działa na Wirtualnej Maszynie Javy i jest kompatybilna z programami napisanymi w Javie
- Łączy programowanie obiektowe i funkcjonalne w jednym zwięzłym języku wysokiego poziomu
- Typy statyczne pomagają uniknąć (typowych w innych językach programowania) błędów w złożonych aplikacjach

Scala – przykład kodu

```
object abstractTypes extends Application {
  abstract class Buffer {
    type T; val element: T
  }
def newIntBuffer(el: Int) = new Buffer {
    type T = Int; val element = el
  }
  def newIntBuffer(el: Int*) = new SeqBuffer {
    type T = Int; val element = el
  }
  println(newIntBuffer(1, 2, 3).length)
}
```

Dla kontrastu – język MTT, którym posługują się dwie osoby na świecie…

- MTT został opracowany w 2019 r. przez absolwenta Politechniki Gdańskiej i z powodzeniem wykorzystany w kilku projektach z dziedziny SI
- Służy do anotacji danych, zastępując linuksową komendę `grep`

```
LABEL a"prawidlowa" @"SERCE" AS @"SERCE W NORMIE"
LABEL @"SERCE" (_|) (ar"powiekszon(e|a)"|a"wieksze") AS @"SERCE POWIEKSZONE"
LABEL @"SERCE" (_|) a"podparte" a"na" a"przeponie" AS @"SERCE PODPARTE NA PRZEPONIE"
LABEL (_|) ar"powiekszon(e|a)" @"SERCE" AS @"SERCE POWIEKSZONE"
LABEL @"SERCE" (a"nie"| ar"trudn(e|a)") a"do" a"oceny" AS @"SERCE NIE DO OCENY"
LABEL a"rtg" AS @"RTG"
```

Python jako nasz główny wybór

Celem szczegółowym przedmiotu jest przekazanie Państwu solidnej warsztatowo podbudowy teoretycznej i praktycznej związanej z zaawansowanym wykorzystaniem **języka Python** do programowania algorytmów sztucznej inteligencji

Kolejnym celem jest nauczenie Państwa (krytycznego) wykorzystywania różnych bibliotek i narzędzi, przydatnych w codziennej pracy programisty SI

Efektywna i wygodna praca programisty – podstawowe zasady

- Indywidualne środowisko pracy, odpowiednio skonfigurowane i odseparowane od reszty systemu
  - Środowisko wirtualne – `venv`
- Narzędzia do pracy grupowej, pozwalające m.in. na efektywne zarządzanie konfiguracją oprogramowania, scalanie zmian pochodzących od wielu osób itp.
  - Systemy kontroli wersji, np. `git`
  - Systemy do śledzenia usterek (bugtrackery), np. w formie hostowanej aplikacji – `GitHub` / `GitLab`
- Konteneryzacja aplikacji (wirtualizacja na poziomie systemu operacyjnego)
  - narzędzia: Docker, Containerd, CRI-O

Python – środowisko wirtualne

- Aplikacje napisane w Pythonie często używają pakietów i modułów, które nie są częścią standardowej biblioteki. Aplikacje te czasami wymagają określonej wersji biblioteki
- Może się zdarzyć, że jedna aplikacja wymaga modułu X w wersji 1.0, a inna w wersji 1.1
- Rozwiązaniem tego problemu jest utworzenie środowiska wirtualnego
- Środowisko takie zawiera instalację języka Python dla określonej wersji języka oraz szereg dodatkowych pakietów w określonych wersjach
  - Instalacja pakietu `venv` dla Pythona (Debian/Ubuntu):
    `apt install pythonX.Y-venv`

Python – środowisko wirtualne – tworzenie i aktywacja – `venv`

- Utworzenie środowiska wirtualnego w wybranym katalogu

```
> python3 -m venv [nazwa_środowiska]
```

- Powstanie katalog `nazwa_środowiska` z językiem Python, biblioteką standardową itp.
- Aktywacja środowiska (Windows / Linux)

```
> [nazwa_środowiska]\Scripts\activate.bat
> source [nazwa_środowiska]/bin/activate
```

- Od tej pory używamy izolowanego środowiska wirtualnego
- Deaktywacja środowiska: `> deactivate`

(znak > oznacza symboliczny „znak zachęty” systemu operacyjnego w trybie linii komend)

Zarządzanie konfiguracją oprogramowania (kontrola wersji)

- Zestaw czynności, pozwalających kontrolować zmiany, jakie mają miejsce w projektach informatycznych
- W szczególności istotne w projektach prowadzonych metodyką zwinną (Agile) – a więc w większości projektów z dziedziny SI
- Realizowane jest to poprzez identyfikację elementów, które mogą się zmieniać, ustalenie relacji pomiędzy nimi oraz określenie mechanizmów zarządzania wersjami
- Odpowiednie zarządzanie konfiguracją jest niezbędne do efektywnej pracy zespołowej

Zarządzanie konfiguracją oprogramowania – scenariusz

- Kolejnym zagadnieniem jest równoległa praca wielu osób nad jednym artefaktem (np. nad jednym plikiem źródłowym, testem, diagramem itp.)
- System zarządzania konfiguracją musi wiedzieć, w jaki sposób pobrać zmiany od poszczególnych programistów, następnie je scalić w jedno, a spójną wersję rozpropagować dalej do pozostałych osób
- Dodatkowo niezbędna jest możliwość śledzenia wszystkich zmian w artefaktach projektu, czyli informacja kto, kiedy i jaką zmianę wprowadził

Systemy do zarządzania konfiguracją – ogólna zasada działania

- Niemal każdy system działa na podobnej zasadzie – za pomocą odpowiednich komend (zwykle wydawanych z konsoli / linii komend) umożliwiają wprowadzanie zmian do repozytorium, pamiętają zmiany artefaktów, umożliwiają synchronizowanie wersji wytworzonych przez różne osoby, a także tworzenie i scalanie gałęzi
- UWAGA! Sam system to nie wszystko – potrzebne są też procedury określające w jaki sposób korzystać z danego narzędzia, np. nazewnictwo wersji, gałęzi, procedury scalania gałęzi itp.

Wzorce zarządzania konfiguracją

- Główna linia (master / main) – gałąź bazowa, w której odbywają się główne prace implementacyjne. Wszystkie zmiany w gałęziach pobocznych powinny być docelowo scalone z gałęzią bazową
- Linia wydania: rozgałęzienie (nowa gałąź), towarzyszące każdemu wydaniu kolejnej wersji systemu / programu. Dzięki temu część funkcjonalności, która była zawarta w tym wydaniu jest odseparowana i można na niej prowadzić równolegle pracę. Jest to niezbędne jeżeli chcemy pozwolić na tworzenie nowej funkcjonalności w głównej gałęzi, a jednocześnie poprawiać błędy w poprzedniej wersji systemu
- Gałęzie dla zadań: gałęzie dla dłuższych zadań, które mogłyby zaburzyć podstawowy kod – na przykład testowanie nowego algorytmu, biblioteki itp.

Najpopularniejsze systemy kontroli wersji

- Git
  - chyba najpopularniejszy
  - działa „out of the box”
  - darmowy hosting (GitHub, GitLab)
- Mercurial
  - też bardzo popularny
  - nieco łatwiejszy do nauczenia
- Subversion (SVN)
  - jeden ze starszych, ale w niektórych projektach jeszcze wykorzystywany
  - scentralizowany

Git – podstawowe komendy (1)

- Inicjalizacja projektu w bieżącym katalogu…

```
> git init
```

… albo sklonowanie istniejącego repozytorium

```
> git clone git@gitlab.com:[konto]/[projekt].git
```

- Rozpoczynamy pracę – najlepiej w nowej gałęzi

```
> git checkout -b [nazwa_gałęzi]
```

- Piszemy kod lub tworzymy inne artefakty
- Dodajemy zmienione pliki do listy plików, które mają być zatwierdzone jako zmienione („staging”)

```
> git add .
```

Git – podstawowe komendy (2)

- Zatwierdzenie („zakomitowanie”) zmian

```
> git commit -m „[opis_zmian]”
```

Przykład: jeżeli używamy systemu typu obsługi zgłoszeń typu „Ticket/issue tracking” (np. w serwisie GitLab / GitHub) i chcemy commitem zamknąć zgłoszenia numer 1 i 2, to piszemy:

```
> git commit -m "Fixes #1, Closes #2"
```

- Wysyłamy nową gałąź do zdalnego repozytorium

```
> git push -u origin [nazwa_gałęzi]
```

Git – podstawowe komendy (3)

- W zdalnym repozytorium tworzymy „merge request” / „pull request” (GitLab / GitHub) – żądanie scalenia naszej nowej gałęzi do gałęzi głównej (*master* / *main*)
- W zależności od zwyczajów w naszej organizacji, następuje dyskusja i ew. głosowanie dot. ew. scalenia naszej gałęzi
- Przełączenie się na gałąź *master* i jej uaktualnienie

```
> git checkout master
> git pull origin master
```

Uwaga! Obecnie promowane jest stosowanie nazwy *main* a nie *master* dla gałęzi głównej

Git – podstawowe komendy (4)

- Pobranie listy zdalnych gałęzi

```
> git fetch --all
> git fetch origin [gałąź_zdalna]
```

- Przełączenie się na pobraną gałąź

```
> git checkout [gałąź_zdalna]
```

Ignorowanie wybranych artefaktów – plik `.gitignore`, np. o zawartości:

```
__pycache__/
venv/
*.py[cod]
```

Git – przydatne „tricki”

- Cofnięcie wszystkich lokalnych zmian

```
> git reset --hard
```

- Zapamiętanie zmian lokalnych w schowku, pobranie zmian zdalnych i ponowne przywrócenie zmian lokalnych ze schowka

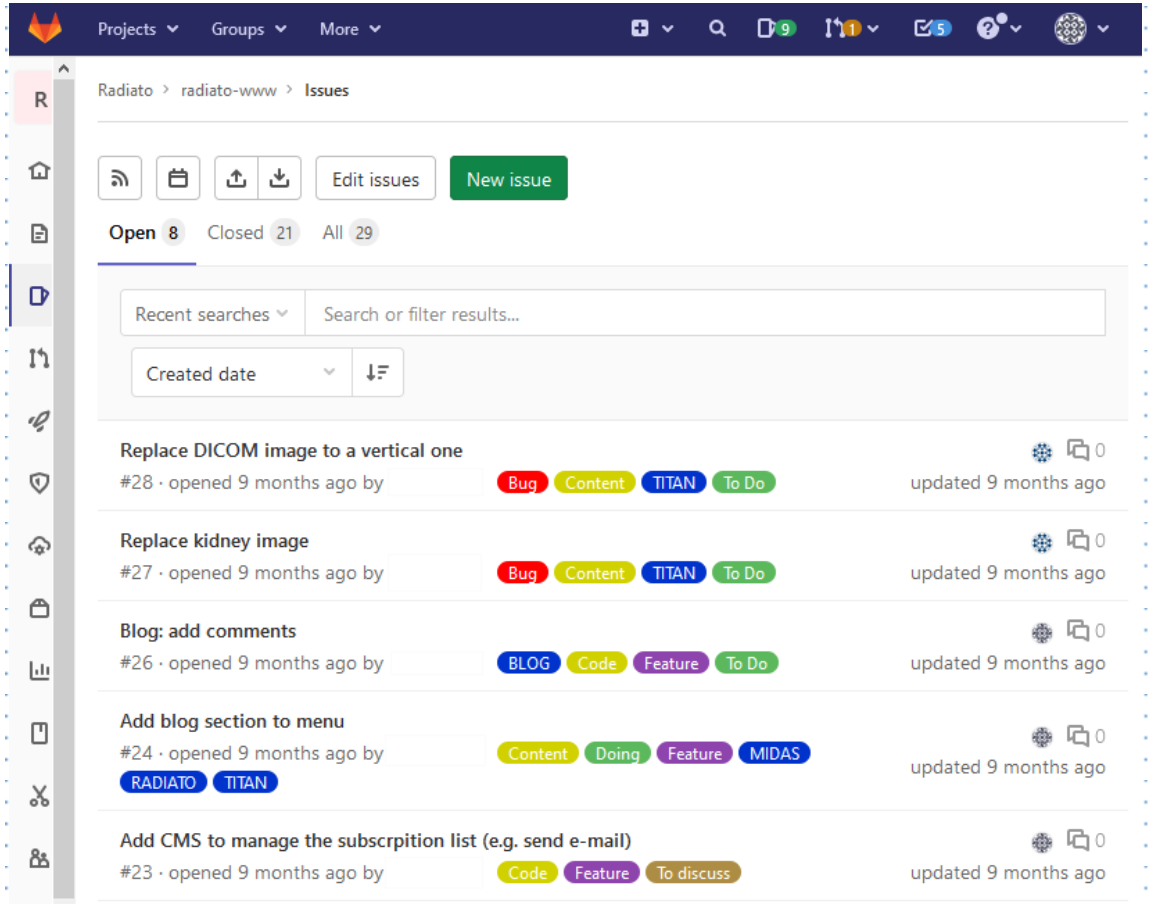
```
> git stash
> git pull
> git stash pop
```

Więcej informacji / dokumentacja git-a

https://git-scm.com/

GitLab jako system śledzenia usterek / zadań do wykonania

Platforma umożliwia łatwe dodawanie usterek (issues), ich etykietowanie, zarządzanie nimi, przypisywanie członkom zespołu do realizacji, zamykanie z linii komend systemu `git` itp.



Konteneryzacja

- „Lżejsza” alternatywa dla wirtualizacji
- Najpopularniejsze narzędzie: Docker
- W kontenerze „aktywujemy” dodatkowy, odizolowany system operacyjny z gotową do działania aplikacją
- Kontener nie emuluje w pełni warstwy sprzętowej – otrzymujemy pełnoprawny system operacyjny wraz ze zdefiniowanym (przez programistę) procesem aplikacji
- Kontenery Dockera działają niezależnie od siebie, realizując pojedynczą funkcjonalność
- Kontenery mogą się ze sobą komunikować za pośrednictwem (wirtualnego) połączenia sieciowego
- Środowisko deweloperskie można zbudować z gotowych obrazów zawierających zainstalowane kontenery i odpowiednie usługi
- Można też dostarczać naszą aplikację w postaci obrazu