Spis treści

1. Wstęp

W ostatnich latach uczenie maszynowe stało się potężnym narzędziem o szerokim zakresie zastosowań, od rozpoznawania obrazów i przetwarzania języka naturalnego do podejmowania decyzji i dokonywania predykcji. Jednym z takich zastosowań jest rozwój algorytmów drzew decyzyjnych, które mogą być wykorzystywane do modelowania i przewidywania wyników na podstawie zbioru danych wejściowych. W niniejszej pracy przedstawiona zostanie aplikacja oparta na drzewach decyzyjnych, która wykorzystuje iteracyjne i odgórne podejście do generowania drzewa decyzyjnego i przewidywania osoby, o której myśli użytkownik. Aplikacja została zaprojektowana tak, aby była responsywna i czytelna dla użytkownika, wykorzystując interfejs internetowy zbudowany przy użyciu React. Algorytm drzewa decyzyjnego jest zaimplementowany przy użyciu Pythona i jest szkolony na zbiorze danych osób i ich powiązanych atrybutów. Wynikowe drzewo decyzyjne może dokładnie przewidzieć osobę, o której myśli użytkownik, zadając serię pytań tak-lub-nie. Niniejsza praca licencjacka opisuje projekt, rozwój i ocenę aplikacji opartej na drzewie decyzyjnym, a także jej potencjalne zastosowania i przyszłe kierunki. Ogólnie rzecz biorąc, ta praca dyplomowa pokazuje moc i wszechstronność algorytmów drzew decyzyjnych w uczeniu maszynowym i ich potencjał dla praktycznych zastosowań w różnych dziedzinach.

2. Dane wejściowe aplikacji

2.1. Źródło danych wejściowych aplikacji

Do szkolenia i testowania algorytmu drzewa decyzyjnego w mojej aplikacji użyte zostały dane z serwisu Wikidata. Serwis ten działa jako centralne miejsce przechowywania ustrukturyzowanych danych siostrzanych projektów Wikimedia, takich jak Wikipedia, Wikivoyage, Wiktionary, Wikiource i inne [1]. Najlepszym sposobem pobrania danych z tej bazy jest Wikidata Query Service, dzięki której można otrzymać konkretne informacje wykorzystując zapytania napisane w języku SPARQL. Ze względu na rozmiar danych których chcieliśmy użyć w aplikacji i limity nałożone w celu uniknięcia przeciążenia bazy, pozyskanie danych osób za pomocą tej usługi było niemożliwe. Aby obejść ten problem, musieliśmy pobrać zrzut bazy danych w formacie JSON i wyodrębnić z niego potrzebne nam rekordy [2]. Wykorzystany w aplikacji zrzut bazy WikiData był wykonany dnia 26 października 2022 roku, więc dane osobowe będą odpowiadać danym aktualnym do tego dnia.

2.2. Struktura danych wejściowych

Wikidata używa strukturalnego modelu danych do przechowywania informacji o encjach, gdzie każda encja jest reprezentowana jako zbiór **deklaracji**. Każda deklaracja składa się z podmiotu, predykatu (właściwości) i obiektu, gdzie podmiot jest opisywaną encją, predykat jest właściwością opisującą podmiot, a obiekt jest wartością właściwości [3]. Szczegółowy podział struktury danych używanych do przechowywania encji w Wikidata wygląda następująco:

1. **ID encji (Entity ID)**: Każda encja w Wikidata jest identyfikowana przez unikalny identyfikator, który jest ciągiem alfanumerycznym zaczynającym się od litery "Q". ID encji jest używane do odróżnienia jednej encji od drugiej.

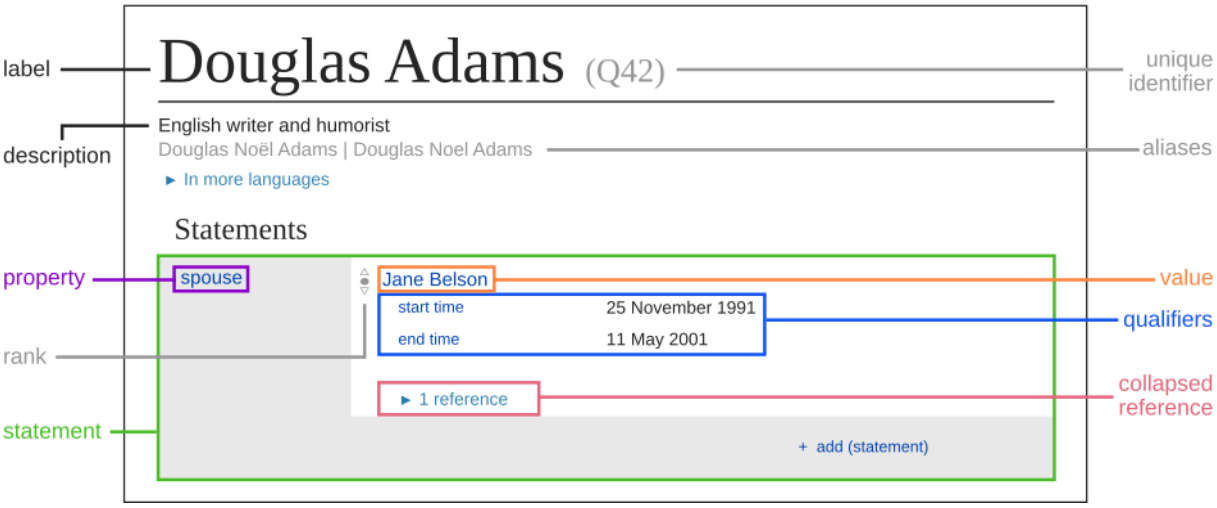
2. **Etykiety, opisy i aliasy**: Encje Wikidata mogą mieć jedną lub więcej etykiet, opisów i aliasów z nimi związanych. Etykiety są nazwami, przez które encja jest znana, podczas gdy opisy dostarczają dodatkowego kontekstu o encji. Aliasy są alternatywnymi nazwami dla encji, które mogą być użyte do odniesienia się do niej.

3. **Odnośniki (Sitelinks**): Odnośniki to linki do odpowiednich stron dla danej encji w innych projektach Wikimedia, takich jak Wikipedia. Odnośniki pozwalają na łatwą nawigację pomiędzy różnymi stronami w sieci, które odnoszą się do tego samego podmiotu.

4. **Identyfikatory**: Identyfikatory to unikalne kody lub numery, które są przypisane do podmiotu przez zewnętrzne organizacje lub bazy danych. Przykłady identyfikatorów obejmują *International Standard Book Numbers* (ISBN), *International Standard Name Identifiers* (ISNI) oraz *Library of Congress Control Numbers* (LCCNs).

5. **Deklaracje (Statements)**: Deklaracje składają się z jednego lub więcej twierdzeń, które opisują podmiot w sposób bardziej szczegółowy. Oświadczenia mogą być traktowane jako zbiór atrybutów podmiotu, gdzie każdy atrybut jest twierdzeniem.

6. **Snaks:** „Snak" jest używany do przechowywania wartości dla własności opisującej daną encję. Każdy „snak" składa się z trzech komponentów: *ID właściwości (Property ID),* który identyfikuje typ reprezentowanych danych, *wartość danych (data value),* która jest rzeczywistą wartością właściwości dla encji i *typ danych (data type),* który określa format wartości danych. Każda deklaracja może zawierać wiele „snaków" dla tej samej właściwości. Pozwala to na reprezentację złożonych struktur danych, takich jak listy lub hierarchie, w ramach modelu danych. Dodatkowo „snaki" mogą posiadać kwalifikatory, które zapewniają dodatkowy kontekst dla wartości danych, np. określając czasowy lub przestrzenny zakres wartości.



Rys. 1 Wizualne przedstawienie modelu struktury danych encji

Źródło:
https://simple.wikipedia.org/wiki/File:Graphic_representing_the_datamodel_in_Wikidata.svg
[dostęp z dn. 14 kwietnia 2023r.]

3. Wyodrębnianie danych ze zrzutu bazy

Na potrzeby naszej aplikacji będziemy potrzebowali wyłącznie informacji dotyczących osób i innych obiektów użytych do ich opisania. Wszystkie encje dotyczące ludzi posiadają właściwość P31 (*instance of)*, dla której przybierają wartość Q5 (*human*). Przeszukanie zrzutu bazy i wybranie encji posiadających te twierdzenie pozwoli nam odizolować interesujące nas dane.

Do wyodrębnienia danych wspomożemy się biblioteką *qwikidata* [4] napisaną w języku programowania Python. Dla naszych potrzeb zaimplementujemy 2 klasy z tej biblioteki:

1. **WikidataItem**: klasa służąca do reprezentacji encji, ułatwiająca wydobywanie informacji z odpowiednich jej elementów.

2. **WikidataJsonDump**: za pomocą tej klasy jesteśmy w stanie stworzyć iterator bezpośrednio ze skompresowanego pliku zrzutu bazy danych, dzięki czemu nie musimy rozpakowywać i ładować całego pliku do pamięci systemowej.

Filtrowanie encji odbywa się za pomocą funkcji *instanceof_human*, zwracającej wartość prawdziwą, jeżeli podana na wejściu encja dla właściwości P31 przyjmuje wartość Q5. Kod źródłowy tej funkcji prezentuje się następująco:

```
P_INSTANCEOF = "P31"
Q_HUMAN = "Q5"

def instanceof_human(item: WikidataItem,
                     truthy: bool = True) -> bool:
    if truthy:
        claim_group = item.get_truthy_claim_group(P_INSTANCEOF)
    else:
        claim_group = item.get_claim_group(P_INSTANCEOF)

    instanceof_qids = [
        claim.mainsnak.datavalue.value["id"]
```

```
        for claim in claim_group
        if claim.mainsnak.snaktype == "value"
    ]
    return Q_HUMAN in instanceof_qids
```

Listing 1. Kod źródłowy funkcji filtrującej encje, które dotyczą osób.

Z pomocą powyższej funkcji można dokonać przeszukania zrzutu bazy. Podczas iteracji po poszczególnych obiektach będą interesowały nas te, które posiadają typ „item" i spełniają warunek P31 = Q5. Przed wyodrębnieniem i zapisaniem tych encji do pliku, poddane one zostaną formatowaniu w celu pozbycia się niepotrzebnych informacji. Właściwości encji staną się listą obiektów, które posiadają 2 pola:

1. **Datatype**: datatype określa typ danych użyty dla wartości danego atrybutu. Ze względu na użyty algorytm i sposób budowania drzewa decyzyjnego zachowane zostaną tylko atrybuty z wartościami tekstowymi. Wartości tekstowe w serwisie Wikidata są wykorzystywane przez typy *wikibase-item, string* i *monolingualtext.* Atrybuty, których wartości są określone innymi typami nie zostały uwzględnione.
2. **Datavalue:** datavalue jest wartością danego atrybutu, w naszym przypadku zawsze to będzie wartość tekstowa. Dla typu *wikibase-item* będzie zawierała ID encji określającej ten atrybut, natomiast w pozostałych przypadkach będzie to inny, dowolny ciąg znaków.

Każdą encja zostanie zapisana jako obiekt o 3 atrybutach: **id**, zawierające ID encji, **label**, zawierającej etykietę encji w języku angielskim, oraz **props**, zawierającej listę atrybutów określających encję. Powyższe operacje obsługiwane są za pomocą pętli przedstawionej poniżej:

```
for entity_dict in wjd_iterator:
    entity_dict = entity_dict.rsplit(",", 1)
    # len = 1 gdy mamy do czynienia z początkiem listy
    if len(entity_dict) == 1:
        continue
```

```
entity_dict = json.loads(entity_dict[0])

if entity_dict["type"] == "item":
    entity = WikidataItem(entity_dict)
    if instanceof_human(entity):
        claims = entity.get_truthy_claim_groups()
        props = {}
        for key in claims.keys():
            ind = claims[key]._claims
            claims_formatted = []
            for claim in ind:
                snak_datatype = claim.mainsnak.snak_datatype
                if snak_datatype not in snak_datatypes
               or claim.mainsnak.snaktype != "value":
                    continue
                else:
                    claims_formatted.append({
                       "datatype": snak_datatype,
                    "datavalue":
                    claim.mainsnak.datavalue.value
                    })
            if len(claims_formatted) > 0:
                props[key] = claims_formatted

        entry = {
            "id": entity.entity_id,
            "label": entity.get_label("en"),
            "claims": props
        }
        humans.append(entry)
```

Listing 2. Kod źródłowy pętli filtrującej, formatującej i zapisującej encje osób

Obiekty te są zapisywane w jednej liście, z której to zostają zapisane do zewnętrznych plików w formacie JSON. Ze względu na ilość osób dostępnych w bazie (około 10 milionów, stan z 26.10.2022r.), lista jest dzielona i zapisywana jako fragmenty po 100 000 obiektów. Taki podział ułatwia zarządzanie danymi i pozwala dokonywać testów pozostałych procesów na mniejszą skalę, bez obciążającego system ładowania wszystkich osób przy każdym podejściu. Generowane pliki zapisywane są w folderze *./resources/humans**n_m**,* gdzie **n** oznacza „numer seryjny" plików (domyślnie 0), a **m** oznacza numer pliku wygenerowanego podczas działania programu. Program ten, ze względu na jednowątkowość jest czasowo zachłanny. Jedną z dostępnych optymalizacji może być podzielenie zrzutu bazy na przedziały. Znając ilość encji w zrzucie bazy, możemy uruchomić kilka instancji programu równolegle, przyznając każdej instancji inny obszar do przeszukania. Dobierając odpowiedni „numer seryjny" plików dla każdej instancji zapewnimy integralność plików (pliki wygenerowane przez jedną instancję nie zostaną nadpisane plikiem wygenerowanym z innej instancji).

Atrybuty typu *wikibase-item* będą zapisane jako ID encji, do której odnosi się dana wartość. Użytkownik końcowy aplikacji nie będzie w stanie rozszyfrować obiektu kryjącego się pod danym identyfikatorem – istnieje więc potrzeba dodania etykiet do takich wartości w celu ich zidentyfikowania. Wykorzystując wygenerowane wcześniej pliki osób jesteśmy w stanie zagregować występujące w nich encje używane jako wartości atrybutów, a następnie znaleźć odpowiadające im etykiety w zrzucie bazy. Lista identyfikatorów używanych jako wartości atrybutów encji osób znajdują się w pliku *propEntities.json**,*** który zostaje wygenerowany za pomocą następującego fragmentu kodu:

```
propEntities = set([])
propertyList = set([])
    print("PropEntities not found. Creating new file...")
    mypath = "../resources/humans"
    filenames = next(walk(mypath), (None, None, []))[2]

    for filename in filenames:
        print(filename)
        with open(mypath + "/" + filename) as f:
            entities = json.loads(f.read())
            for i in entities:
                for claim in i['claims']:

                    if claim not in propertyList:
```

```
                    propertyList.add(claim)

                for value in i['claims'][claim]:
                    if value['datatype'] == 'wikibase-item':
                        if value['datavalue']['id'] not in propEntities:
                            propEntities.add(value['datavalue']['id'])
    print(f"done {filename} ", len(propEntities), len(propertyList))

with open("../resources/propEntities/propEntities.json", "w") as f:
    f.write(json.dumps(list(propEntities)))

with open("../resources/itemProps/humanItemProps.json", "w") as f:
    f.write(json.dumps(list(propertyList)))
```

Listing 3. Fragment kodu źródłowego generujący pliki z używanymi identyfikatorami obiektów oraz atrybutów

W trakcie działania powyższego programu zostanie również wygenerowany plik *humanItemProps.json*, zawierający listę identyfikatorów atrybutów użytych do opisu encji osób – będzie on potrzebny w dalszej części przy zapisie rodzajów atrybutów do bazy i przy tworzeniu drzewa decyzyjnego.

Proces wyodrębniania etykiet z zrzutu bazy jest analogiczny do procesu wyodrębniania encji osób i ich właściwości. Encje są iteratywnie odnajdywane w pliku zrzutu, formatowane do obiektów o 2 polach – **id i label**, i zapisywane do list długości 100 000. Pliki generowane są pod ścieżką *./resources/propValues/propValues**n_m***, gdzie **n** oznacza „numer seryjny” generowanych plików, a **m** oznacza numer pliku wygenerowanego przez program. W celach optymalizacyjnych można zastosować ten sam proces co przy wyodrębnianiu encji osób – należy uruchomić kilka instancji programu równocześnie, dostosowując odpowiednio obszary przeszukiwań i „numer seryjny” generowanych plików.

4. Eksport danych do bazy PostgreSQL

Przechowywanie tak dużej ilości danych w plikach może przysparzać wielu problemów, zarówno optymalizacyjnych jak i tych związanych z bezpieczeństwem. Dzisiejszym de facto standardem radzącym sobie z tymi problemami jest używanie systemu bazodanowego. System bazy danych jest systemem oprogramowania przeznaczonym do przechowywania, zarządzania i wyszukiwania dużych ilości danych strukturalnych. Zapewnia on scentralizowane repozytorium danych, które może być dostępne dla wielu użytkowników i aplikacji, zwykle zawierający narzędzia do modelowania, wprowadzania, wyszukiwania i analizy danych [5]. W tej aplikacji wykorzystana została relacyjna baza danych PostgreSQL – wysoce skalowalny system open-source wykorzystujący język SQL do obsługi systemu i tworzenia zapytań. Serwer bazodanowy Postgres będzie działał w kontenerze systemu Docker. Niesie to ze sobą następujące benefity:

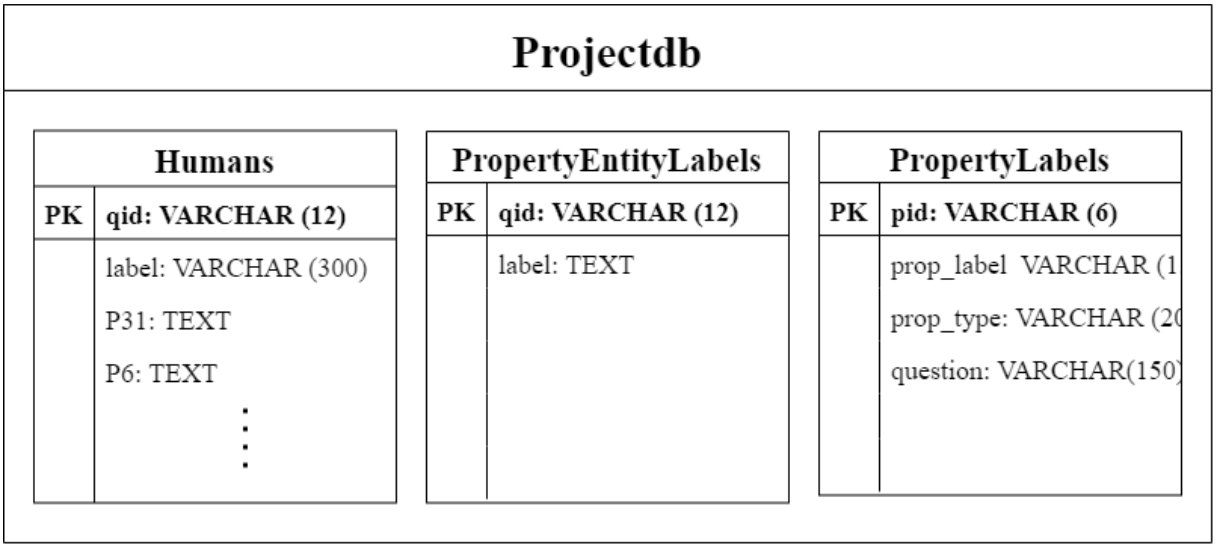
1. Przenośność: Docker pozwala nam spakować naszą aplikację i bazę danych w pojedynczy kontener, który może być łatwo wdrożony i uruchomiony na każdej maszynie, która obsługuje Dockera. Ułatwia to przenoszenie naszej aplikacji i bazy danych pomiędzy środowiskami developerskimi, testowymi czy produkcyjnymi.

2. Izolacja: Kontenery Docker zapewniają wysoki stopień izolacji między naszą aplikacją a systemem operacyjnym maszyny, co pomaga zapobiegać konfliktom i zapewnia, że nasza aplikacja działa spójnie w różnych środowiskach.

3. Odtwarzalność: Docker pozwala nam stworzyć spójne i powtarzalne środowisko dla naszej aplikacji i bazy danych. Pomaga to zapewnić, że nasza aplikacja działa w ten sam sposób na różnych maszynach i środowiskach, a także ułatwia debugowanie i rozwiązywanie problemów.

4. Łatwość wdrażania: Docker ułatwia wdrożenie naszej aplikacji i bazy danych poprzez zapewnienie standardowego sposobu pakowania i dystrybucji naszego kontenera. Zmniejsza to ryzyko błędów i upraszcza proces wdrażania, co pozwala na zaoszczędzenie czasu i wysiłku.

W bazie danych zostaną utworzone 3 tabele do przechowywania danych: **Humans**, przechowująca dane osób, **PropertyEntityLabels**, przechowująca etykiety obiektów opisujących właściwości osób, oraz **PropertyLabels**, przechowująca etykiety atrybutów, typy atrybutów i szablony pytań, które będą wykorzystywane w dalszej części aplikacji. Schemat struktury bazy jest przedstawiony na poniższym wykresie:



Rys. 2 Strukura bazy danych Projectdb

Pole *question* tabeli *PropertyLabels* zawiera ręcznie napisane pytania do wyselekcjonowanych atrybutów w języku angielskim. Będą one stanowiły szablon dla pytań stawianych użytkownikowi korzystającemu z aplikacji. Przykładowo, dla atrybutu o PID p17 (country – kraj pochodzenia), pole *question* posiada wartość „Is this person from QQ ?”, gdzie „QQ” będzie zastępowane optymalną wartością w danym stanie gry, dostarczoną przez wygenerowane drzewo decyzyjne (np. pytanie „Is this person from QQ ?” zostanie w aplikacji wyświetlone jako „Is this person from Germany ?”).

Standardową metodą dodawania rekordów do tabel w języku SQL jest komenda **INSERT INTO**, za pomocą której można dodać maks. 1000 rekordów przy użyciu jednego zapytania. W naszym przypadku, chcąc dodać miliony rekordów, bylibyśmy zmuszeni wykonać tysiące zapytań INSERT INTO, co wiązałoby się z dużymi obciążeniami po stronie serwera bazodanowego. Jeśli chodzi o wstawianie dużych ilości rekordów do bazy danych PostgreSQL, użycie zapytania **COPY** jest ogólnie uważane za szybsze i bardziej wydajne niż użycie zapytania INSERT. Głównym powodem tego jest fakt, że zapytanie COPY zostało

zaprojektowane do obsługi dużych ilości danych i jest zoptymalizowane do ich masowego ładowania. Dzieje się tak, ponieważ zapytanie COPY omija kilka etapów przetwarzania, które są zaangażowane w pojedyncze zapytania INSERT, takie jak parsowanie i planowanie zapytania, przetwarzanie planu zapytania oraz sprawdzanie ograniczeń i wyzwalaczy. Zamiast tego zapytanie COPY odczytuje dane bezpośrednio z pliku i wstawia je do bazy danych, co zmniejsza narzut związany z poszczególnymi zapytaniami [6].

Warto jednak zauważyć, że zapytanie COPY nadaje się tylko do ładowania danych z pliku, podczas gdy zapytanie INSERT może być używane do wstawiania danych programowo lub interaktywnie. Ponadto zapytanie COPY wymaga, aby dane były w określonym formacie, takim jak CSV lub TSV, co będzie wymagało dodatkowego przetwarzania danych.

Chcąc przekonwertować dane osób z pliku JSON do pliku CSV, trzeba było rozwiązać następujące problemy:

1. W wielu przypadkach ma miejsce sytuacja, gdzie dla danej osoby dla jednego atrybutu przypisanych zostaje wiele wartości, np. jedna osoba mogła pełnić wiele zawodów lub mieć wiele rodzeństwa. Fakt, że w naszej bazie jeden atrybut może mieć wyłącznie jedną wartość tekstową, zmusza nas do wykonania pewnej konwersji. Wartości tekstowe lub QID encji reprezentujących wartości danego atrybutu zostaną połączone ze sobą w jeden ciąg znaków, a poszczególne wartości będą oddzielone znakiem pionowej belki (**|**) w celu późniejszego ich rozróżnienia.

2. Niektóre wartości będą zawierały w sobie znak przecinka, który jest domyślnym separatorem używanym w plikach CSV. Z tego względu jako separator zostanie użyty znak średnika (**;**), a ewentualne średniki istniejące w wartościach tekstowych zostaną usunięte.

3. Zdecydowana większość osób dla wielu atrybutów nie będzie posiadała wartości; muszą one jednak być uwzględnione, aby import danych do tabeli odbył się poprawnie. Jeśli dla danej encji w pliku znajdą się dwa średniki obok siebie (**;;**), system PostgreSQL dla danego z kolei atrybutu wstawi wartość pustą i będzie w stanie kontynuować import. Aby tego dokonać, użyjemy wcześniej wygenerowanego pliku *humanItemProps.json* zawierającego listę wszystkich używanych do opisu ludzi

atrybutów. Każdy atrybut zostanie zmapowany na wartość tego atrybutu dla danej osoby lub średnik w przypadku braku wartości.

Powyższe rozwiązania i sama generacja pliku CSV została zaimplementowana w języku Javascript i wygląda następująco:

```
for (const file of files) {

    let entities = require(`../resources/humans/${file}`)

    let formattedData = entities.map((ent, ind) => {
        const label = ent.label.replace(";", "")
        if (label == '') return ''

        const claimList = humanProps.map((prop, ind) => {
        if (Object.keys(ent['claims']).includes(prop)) {
            const claimValues = ent['claims'][prop].map(value => {

            switch(value['datatype']) {
                case 'monolingualtext':
                return `${value['datavalue']['text'].replace(/;/g, "")}`

                case 'wikibase-item':
                return `${value['datavalue']['id']}`

                case 'string':
                return `${value['datavalue'].replace(/;/g, "")}`

            }
            }).join(" | ")

            return claimValues

        } else return ''
        })
        .join(';')
        if (claimList.split(";").length - 1 !== 845) {
            console.log(claimList.split(";").length - 1, ent['id'], label)
```

```
        }

        const finalString = `${ent['id']};${label.replace(/;/g,
"")};${claimList}\n`

        if (finalString.split(";").length - 1 !== 847) {
            console.log(finalString.split(";").length - 1, ent['id'], label)
        }
        return finalString
    })
        .join('')


    console.log(`Data copied from ${file} into humans.csv`)

    fs.appendFileSync(`./humans.csv`, formattedData)

    entities = null
    formattedData = null
  }
```

Listing 4. Fragment kodu źródłowego transformujący dane osobowe i generujący plik CSV

Dane do tabeli *PropertyEntityLabels* również zostaną przekonwertowane do pliku CSV w celu optymalizacji importu. Ze względu na prostą strukturę nie będą one wymagały takiej transformacji jak dane poprzednie. Jedynym napotkanym problemem było występowanie duplikatów danych źródłowych – encje o zduplikowanym ID nie mogłyby zostać dodane do bazy, zwracając błąd. W tym celu zaimplementowano warunek sprawdzający, czy encja o danym ID jest już dodana do pliku. Fragment kodu transformujący dane encji do pliku CSV wygląda następująco:

```
   let qids = new Set()

   files.forEach((file) => {
       const values = require(`../resources/propValues/${file}`)

       let insertquery = values.map((item, ind) => {
           const value = item['label'].replace(/,/g, " ")
```

```
            if (value != ''){
                if (!qids.has(item['id'])) {
                    qids.add(item['id'])

                    if ((value.split('"').length - 1) % 2 == 0) return
`${item['id']},${value}\n`
                    else return `${item['id']},${value}"\n`
                } else return ''
            } else return ''
        })

        let csv = insertquery.join('')

        fs.appendFile('./entities.csv', csv, () => {
            console.log(`appended from ${file}`)
        })

        insertquery = {}
        csv = ''
    })
```

Listing 5. Fragment kodu źródłowego transformujący dane encji opisujących osoby i generujący plik CSV

Utworzone pliki CSV trzeba przenieść do środowiska serwera bazodanowego, skąd ich dane będą mogły zostać zaimportowane do odpowiednich tabel. Zanim będzie można do tego przejść, należy utworzyć same tabele, które będą przechowywały dane. W przypadku tabeli *Humans* wykorzystany został plik *humanItemProps.json,* zawierający listę PID atrybutów opisujących osoby. Każdy identyfikator z listy został zmapowany na wartość tekstową pozwalającą utworzyć kolumnę o danym PID. Pozostałe tabele zostały stworzone za pomocą prostych zapytań SQL.

Do połączenia z serwerem PostgreSQL z poziomu kodu Javascript wykorzystany został moduł **node-postgres** [7]. Udostępnia on konfigurowalny obiekt *Client*, za pomocą którego można dokonać połączenia z bazą danych Postgres i wykonywać zapytania. Operacje kopiowania plików CSV i importowania danych do tabel mogą zostać wykonane manualnie, zdecydowano jednak aby proces ten również umieścić w tym samym programie w celu zautomatyzowania całego procesu i ułatwienia użytkowania. Do zrealizowania komend z wiersza poleceń wykorzystano moduł **shelljs** [8]. W zależności od konfiguracji maszyny

docker na urządzeniu użytkownika, przy uruchamianiu programu konieczne może być posiadanie uprawnień administratora / roota aby wykonać zawarte polecenia. Poniżej przedstawiono fragment kodu tworzącego tabelę *Humans* i dokonującego importu danych z pliku. Proces ten jest analogiczny dla tabel *PropertyLabels* i *PropertyEntityLabels*:

```
    client
    .connect()
    .then(async () => {
      console.log('Connected to PostgreSQL');

      const createQuery = `CREATE TABLE IF NOT EXISTS Humans (
        qid VARCHAR(12) PRIMARY KEY,
        label VARCHAR(300) NOT NULL,
        ${humanProps.map((key, ind) => {
          const datatype = datatypes[props[key]['type']]
          if (props[key]['type'] == "WikibaseItem") {
            if (ind+1 == Object.keys(props).length) return `${key} ${datatype}`
            else return `${key} ${datatype}`
          } else {
            if (ind+1 == Object.keys(props).length) return `${key} ${datatype}`
            else return `${key} ${datatype}`
          }
        })}
      );
      `

      await client.query(createQuery)
      .then(() => console.log('Humans table created'))
      .catch(err => console.error('Creation error', err.stack))

      if (shell.exec(`docker cp ./humans.csv ${dockername}:/humans.csv`).code !==
0) {
        shell.echo('Error: Unable to copy files to docker container.');
        shell.exit(1);
      }

      console.log("Copied to docker")

      const copyQuery = `COPY Humans FROM '/${file}' DELIMITER ';' quote E'\b'
CSV;`

      client.query(copyQuery)
      .then(() => {
        console.log(`Data from ${file} copied to Humans`)
        shell.rm('-rf', 'humans.csv')
```

```
    exit(0)
  })
  .catch(err => {
    console.error('Insertion error', err.stack)
    exit(1)
  })

})
.catch(err => {
  console.error('Connection error', err.stack)
  exit(1)
})
```

Listing 6. Kod źródłowy tworzący tabelę *Humans* i importujący dane osobowe z pliku *humans.csv*.

5. Tworzenie drzewa decyzyjnego

5.1. Rozważane metody podjęcia problemu

Zamierzonym efektem pracy było stworzenie aplikacji będącej w stanie odgadnąć osobę, o której myśli użytkownik, na podstawie serii pytań tak / nie. Ważne było, aby ilość pytań prowadząca do danej osoby była możliwie jak najmniejsza, nie doprowadzając jednocześnie do długich przerw między pytaniami spowodowanych skomplikowanymi obliczeniami. Pierwszą testowaną metodą było zaimplementowanie algorytmu ID3 [9]. Algorytm ten rekurencyjne dzieli zbiór danych na podstawie atrybutu, który daje największy **zysk informacyjny** (ang. *information gain*). Drzewo budowane jest od góry do dołu, zaczynając od węzła głównego i dodając węzły potomne, gdy dzieli dane w oparciu o wybrany atrybut. Wynikowe drzewo może być użyte do klasyfikacji, gdzie każdy węzeł liścia reprezentuje klasę.

Algorytm ID3 składa się z czterech głównych elementów:

1. Zysk informacyjny (ang. *Information Gain*): Jest to miara tego, jak wiele informacji dostarcza atrybut dla zadania klasyfikacji. Zysk informacyjny jest obliczany poprzez pomiar różnicy w entropii (lub nieczystości) pomiędzy węzłem rodzica a węzłami potomnymi wynikającymi z podziału na atrybut. Atrybut o wysokim zysku informacyjnym jest wybierany jako atrybut podziału.

2. Entropia (ang. *Entropy*): Entropia jest miarą nieczystości zbioru danych. Zbiór danych o niskiej entropii jest homogeniczny, co oznacza, że wszystkie dane należą do tej samej klasy. Zbiór danych z wysoką entropią jest heterogeniczny, co oznacza, że zawiera dane z różnych klas. Entropia jest obliczana jako suma prawdopodobieństwa każdej klasy pomnożona przez logarytm prawdopodobieństwa.

3. Selekcja atrybutów: Jest to proces wyboru najlepszego atrybutu do podziału zbioru danych. Algorytm ID3 wykorzystuje zysk informacyjny, aby wybrać atrybut, który zapewnia największy zysk informacyjny.

4. Budowa drzewa: Po wybraniu najlepszego atrybutu, zbiór danych jest dzielony na podzbiory w oparciu o wartości wybranego atrybutu. Proces ten jest powtarzany dla każdego podzbioru, aż wszystkie instancje w podzbiorze będą należały do tej samej klasy. W tym momencie tworzony jest węzeł liścia reprezentujący klasę i proces ten jest kontynuowany rekurencyjnie.

Implementacja klasycznego algorytmu ID3 i testowanie jego działania na danych testowych wykazało szereg problemów związanych ze specyfiką danych, mianowicie:

1. Algorytm nie był w stanie dobrać optymalnych atrybutów do podziału danych. Można przypuścić, że jest to spowodowane nietypowym zestawem danych, który może być problematyczny dla algorytmów drzew decyzyjnych. Standardowo, model klasyfikacyjny tworzy się w oparciu o kilka wybranych klas ze zbioru szkoleniowego. Algorytm wtedy jest w stanie odnaleźć zależności, na podstawie których będzie w stanie określić, do której klasy należy obiekt. W naszym przypadku każdy obiekt reprezentował inną klasę, więc algorytm nie był w stanie dobrać wartości atrybutów dzięki którym byłby w stanie określić klasę przyszłego, testowanego obiektu.

2. Domyślnie, algorytm ID3 nie obsługuje sytuacji, w której obiekt nie posiada wartości dla wybranego atrybutu. Podział danych ze względu na brak atrybutu może być w niektórych przypadkach bardzo istotny (np. osoby żyjące nie będą posiadały atrybutu z datą śmierci), dlatego warto byłoby zaimplementować obsługę takich sytuacji.

Sam proces budowy struktury drzewa był satysfakcjonujący, dlatego postanowiono zostać przy algorytmie ID3, modyfikując przy tym metodę obliczania zysku informacyjnego i dostosowując algorytm do obsługi wartości pustych.

5.2. Implementacja zmodyfikowanego algorytmu

Właściwy algorytm generujący drzewo został zaimplementowany w języku Python, w którym implementowany był również domyślny algorytm ID3. Główną funkcją odpowiedzialną za tworzenie drzewa jest funkcja *make_tree*. Funkcja ta przyjmuje parametry:

- **root –** obiekt stanowiący korzeń, z którego będzie generowane dalsze drzewo
- **prev_feature_value** – atrybut użyty do podziału danych w poprzedniej iteracji funkcji, przyjmuje wartość pustą *None* dla najwyższego wywołania funkcji
- **train_data** – zbiór danych zawierający klasy i wartości dla wybranych atrybutów
- **class_list** – lista zawierająca wszystkie klasy obecne w zbiorze danych
- **feature_list –** lista wszystkich dostępnych atrybutów

Proces budowania drzewa decyzyjnego wygląda następująco:

1. Na podstawie danych wejściowych *train_data, feature_list* i *class_list* obliczany jest atrybut z największym zyskiem informacyjnym. Jeśli zwrócona zostanie pusta wartość, algorytm kończy działanie.
   1.1. Zysk informacyjny dla każdego atrybutu jest równy liczbie wystąpień danego atrybutu w zbiorze podzielonym przez liczbę unikalnych wartości występujących dla tego atrybutu. Przypadki, w których obiekt posiada wiele wartości dla jednego atrybutu (rozdzielonych znakiem „|”, zgodnie z wcześniej przyjętą konwencją), wszystkie wartości będą odpowiednio sprawdzone pod kątem unikalności.
2. Generowane jest poddrzewo *tree* i nowy zbiór danych *new_train_data* z wykorzystaniem atrybutu z największym zyskiem informacyjnym (*max_info_feature*), *train_data* i *class_list.*
   2.1. Funkcja iteruje po wszystkich elementach listy *class_list*, tworząc obiekt *feature_value_count_dict*, zawierający pary wartość atrybutu – ilość wystąpień danej wartości.
   2.2. Dla każdego pola obiektu *feature_value_count_dict* wykonywane są następujące operacje:
      2.2.1. Tworzy nowy obiekt *feature_value_data*, zawierający encję ze zbioru *train_data*, których wartość dla atrybutu *max_info_feature* jest równa odpowiadającej wartości danej iteracji.

2.2.2. Każda klasa z listy *class_list* jest sprawdzana pod kątem bycia potencjalnym kandydatem na bycie „liściem” (węzłem końcowym, z którego dalsza generacja drzewa nie będzie możliwa). Jeśli dana klasa będzie istnieć we wcześniej utworzonym zbiorze *feature_value_data*, a ilość wystąpień danej wartości atrybutu będzie równa 1, wtedy klasa ta staje się liściem drzewa. Do obiektu *tree* dodawane jest pole o kluczu *feature_value* i o wartości odpowiadającej obecnej klasie. Dodatkowo wszystkie encje ze zbioru *train_data,* dla których wartość atrybutu *feature_name* jest równa wartości wykorzystanej w liściu są usuwane ze zbioru. Dzięki temu przyszłe iteracje generacji drzewa będą mogły zignorować encje, które znalazły już swoje miejsce na drzewie, optymalizując tym samym działanie procesu. W przypadku, gdy żadna klasa nie może zostać liściem, w obiekcie *tree* pole o kluczu *feature_value* otrzymuje wartość „**?**”, sygnalizując przyszłym iteracjom funkcji *make_tree*, że w danym obrębie drzewa istnieje miejsce do dalszego podziału i ekspansji drzewa.

2.3. Obiekt poddrzewa *tree* i nowy, przefiltrowany zbiór danych *new_train_data* zostaje zwrócony do funkcji *make_tree.*

3. Jeśli wartość *prev_feature_value* nie jest pusta, w obiekcie *root* pod kluczem *prev_feature_value* tworzony jest pusty obiekt, w nim z kolei pod kluczem *max_info_feature* wstawiane jest wcześniej wygenerowane drzewo *tree*. Wartość ta zostaje zapisana jako *next_root* do dalszego użytku. Jeżeli wartość *prev_feature_value* jest pusta, wtedy mamy doczynienia z początkiem drzewa decyzyjnego – w obiekcie *root* pod kluczem *max_info_feature* wstawiane jest wygenerowane drzewo *tree*. Wartość ta również zostaje zapisana jako *next_root* do dalszego użytku.
4. Z listy atrybutów *feature_list* usuwany jest atrybut *max_info_feature*, jeżeli jest to możliwe.
5. Dla każdego pola obiektu *next_root* wykonywane są następujące operacje:

5.1. Jeżeli wartość pola jest równa „**?**”, tworzony jest nowy obiekt *feature_value_data*, zawierający encję ze zbioru *train_data*, których wartość dla atrybutu *max_info_feature* jest równa odpowiadającej kluczowi pola danej iteracji. W oparciu o obiekt *feature_value_data* tworzona jest również nowa lista klas *new_class_list*. Następnie wywoływana rekursywnie jest funkcja *make_tree* z następującymi atrybutami:

5.1.1. **root** – obiekt *next_root*,

5.1.2. **prev_feature_value –** odpowiadający danej iteracji klucz pola obiektu *next_root*,

5.1.3. **train_data –** obiekt *feature_value_data*,

5.1.4. **class_list –** lista *new_class_list*,

5.1.5. **feature_list** – ten sam obiekt *feature_list*, z którego został wcześniej atrybut *max_info_feature*.

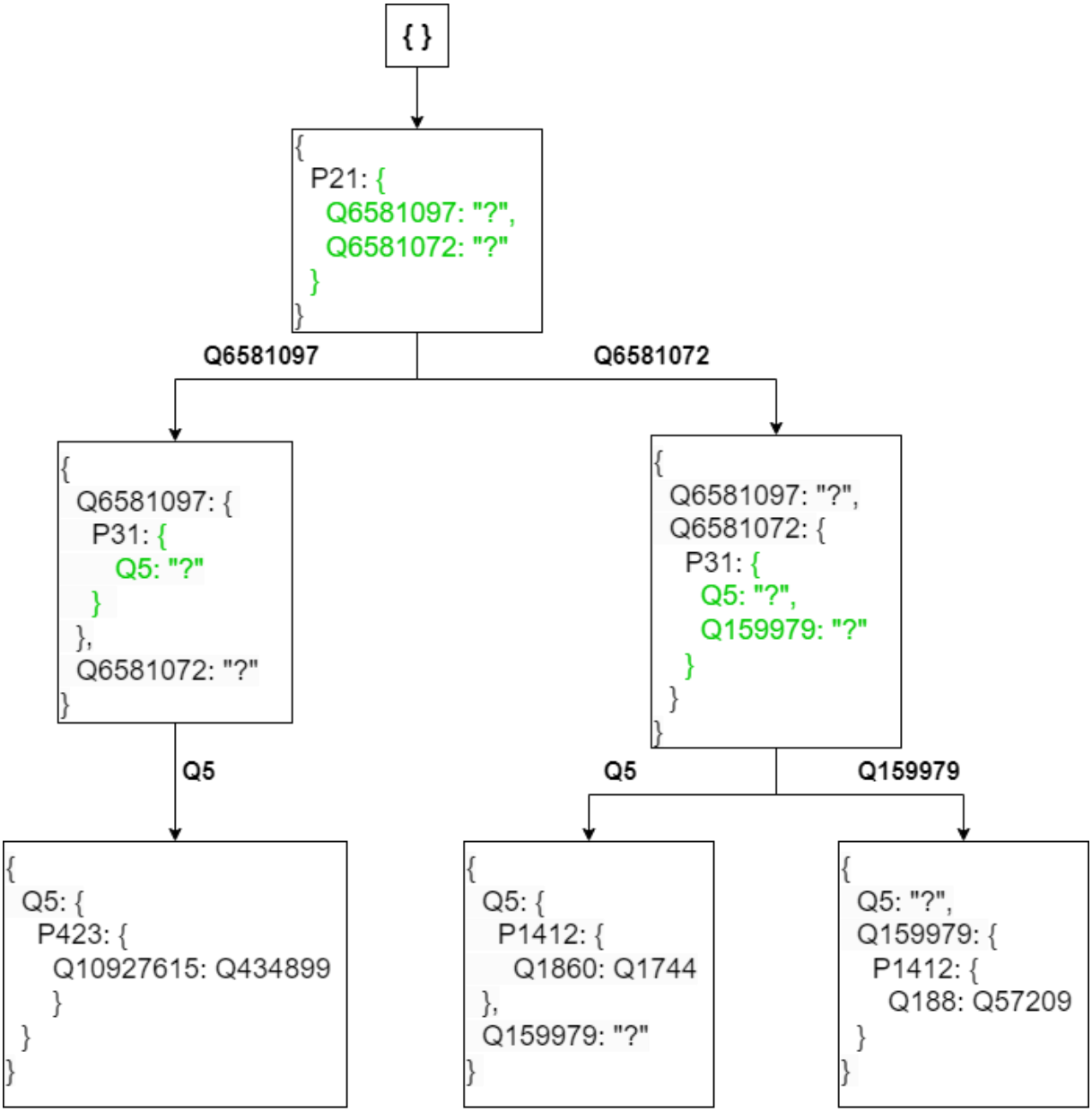
5.2. Jeżeli wartość pola jest różna od „**?**”, pętla iteracyjna przechodzi do następnego elementu.

6. Proces ten zachodzi dopóki ilość elementów w zbiorze danych i w zbiorze nazw atrybutów jest różna od 0.

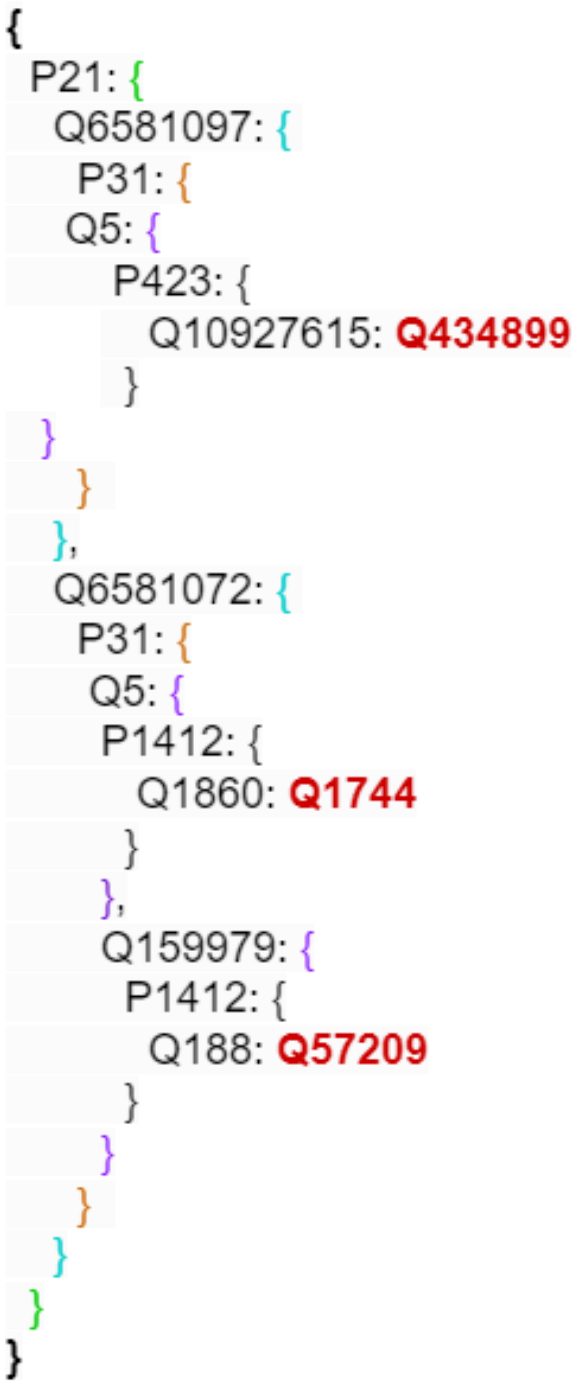
```
from find_most_informative_feature import find_most_informative_feature
from generate_sub_tree import generate_sub_tree


def make_tree(root, prev_feature_value, train_data, class_list, feature_list):
    """Recursively creates a decision tree from a given dataset.

    :param root: A dict from which the tree will be generated.
    :param prev_feature_value: A feature value used in previous iteration. This
should be *None* for
             the top function call.
    :param train_data: A dataset containing classes and their feature values.
    :param class_list: A list of
    :param feature_list: A list of available attributes.
    :return: A dict containing the decision tree.
    """
    if len(train_data.keys()) != 0 and len(feature_list) != 0:
        max_info_feature = find_most_informative_feature(train_data, feature_list,
class_list)
        # ------ Jeśli nie ma atrybutów do wybrania, kończymy działanie funkcji
        if max_info_feature is None:
            return
        tree, new_train_data = generate_sub_tree(max_info_feature, train_data,
                                                 class_list)
        # ------ dołączenie do poprzedniej wartości w drzewie
        if prev_feature_value is not None:
            root[prev_feature_value] = dict()
            root[prev_feature_value][max_info_feature] = tree
            next_root = root[prev_feature_value][max_info_feature]
```

```
        else:
            # ------ dołączenie do korzenia drzewa
            root[max_info_feature] = tree
            next_root = root[max_info_feature]

        feature_list_cp = feature_list.copy()
        try:
            feature_list_cp.remove(max_info_feature)
        except ValueError:
            # "Obsługa" wyjątku
            ignore = 'this'

        for node, branch in list(next_root.items()):  # iterating the tree node
            if branch == "?":  # if it is expandable
                # ------ dodawanie poddrzewa
                if node == "None":
                    feature_value_data = {k: v for k, v in
list(new_train_data.items()) if v.get(max_info_feature, None) is None}
                else:
                    feature_value_data = {k: v for k, v in
list(new_train_data.items()) if isinstance(v.get(max_info_feature, None), str) and
node in v[max_info_feature]}
                new_class_list = list(feature_value_data.keys())
                # ------ rekursywne odwołanie z zbiorem danych
                make_tree(next_root, node, feature_value_data, new_class_list,
feature_list_cp)
```

Listing 7. Kod źródłowy funkcji *make_tree* generującej drzewo decyzyjne



Rys. 3 Przykładowy przebieg budowy drzewa funkcją *make_tree*. Kolorem zielonym oznaczono obiekty, które będą wykorzystane jako *next_root* w kolejnej iteracji.



Rys. 4 Drzewo decyzyjne uzyskane w wyniku procesu przedstawionego na Rys. 3. Kolorem czerwonym oznaczone są węzły końcowe (liście).

Pozyskanie danych potrzebnych do stworzenia drzewa, jak i jego same tworzenie odbywa się za pomocą funkcji *gen_decision_tree*, przyjmującej parametr *num_entries* określającej ilość encji osób pobranych z bazy danych. Domyślną wartością dla tego atrybutu jest 100 000. Tworzenie drzewa decyzyjnego z dużą ilością osób jest procesem bardzo czasochłonnym i wymagającym sporej ilości pamięci systemowej – generacja drzewa dla 1 000 000 na maszynie testowej wymagała zarezerwowania 1.5GB pamięci systemowej i trwała ok. 2 godziny. Warto zauważyć, że osoby pobierane są w rosnącej kolejności ich QID. Osoby o krótszym QID zwykle są tymi bardziej rozpoznawalnymi i bardziej opisanymi – krótki QID wskazuje na wczesne stworzenie wpisu o danej osobie. Dzięki temu nawet przy ograniczonym zbiorze danych znajdzie się wiele osób, które będą znane przez użytkowników, a tym samym będą w stanie trafnie odpowiedzieć na zadane pytania. Z drugiej strony ograniczony zbiór danych prawdopodobnie nie będzie zawierał obecnie popularnych celebrytów / celebrytek, o których kilka lat temu jeszcze nie słyszano, przez co zostali dodani później do bazy Wikidata (np. Khaby Lame – ur. 2000r. senegalski celebryta i osobowość w mediach społecznościowych, QID: Q106839925, lub Olivia Rodrigo – ur. 2003r. amerykańska piosenkarka i aktorka, QID: Q63243883).

Funkcja *gen_decision_tree* łączy się z bazą danych za pomocą modułu **psycopg2** [10]. Moduł ten udostępnia metodę *connect()*, tworzącą instancję połączenia z bazą danych, oraz metodę *cursor()*, pozwalającą na utworzenie **kursora** bazodanowego. Kursor jest nazwanym obiektem bazy danych, który pozwala na pracę z zestawem wierszy zwróconych przez instrukcję SELECT. Koncepcja kursora jest podobna do iteratora w takich językach programowania jak Java czy Python – pozwala on na pobieranie wierszy pojedynczo z zestawu wyników i przetwarzanie ich. Kursory mogą być używane do poprawy wydajności i zmniejszenia zużycia pamięci podczas pracy z dużymi zbiorami danych, ponieważ pozwalają na pobieranie tylko tych danych, które są potrzebne w danym momencie, zamiast pobierać cały zbiór danych na raz. Mogą być one używane do przewijania zbioru wyników do przodu i do tyłu lub do przewijania do określonego wiersza w zbiorze wyników. Kursory mogą być również używane do aktualizacji i usuwania wierszy w tabeli, jak również do zapytania o dane z wielu tabel [11]. W naszym programie utworzone zostaną 2 kursory:

1. Kursor pobierający informację o dostępnych atrybutach za pomocą zapytania SQL:

```
SELECT table_schema, table_name, column_name
                FROM information_schema.columns
                WHERE table_schema = 'public'
                AND table_name = 'humans';
```

2. Kursor pobierający dane osób za pomocą zapytania SQL:

```
SELECT * FROM humans
             ORDER BY length(qid)
             LIMIT {num_entries};
```

   gdzie *num_entries* oznacza wcześniej zadeklarowaną ilość encji, która ma zostać pobrana z bazy.

Dane z kursora 2. są pobierane partiami, aby po ich przetworzeniu mogły zostać usunięte z procesu w celu zmniejszenia zużycia pamięci systemowej. Po utworzeniu obiektu drzewa decyzyjnego wywołaniem funkcji *make_tree*, jest on zapisywany w postaci JSON pod ścieżką *../resources/decision_tree.json*.

6. Struktura aplikacji

Aplikacja została podzielona na 3 zasadnicze komponenty:

1. Serwis bazodanowy REST API – nawiązuje bezpośrednie połączenie z serwerem PostgreSQL, z którego pozyskuje dane w oparciu o parametryzowane żądania HTTP.
2. Serwis obsługi drzewa decyzyjnego – oblicza najlepszy przebieg gry na podstawie wygenerowanego drzewa i udzielonych wcześniej przez użytkownika odpowiedzi. Łączy się z serwisem bazodanowym w celu pobrania niezbędnych informacji, m.in. etykiet encji i szablonów pytań, które są później wysyłane do aplikacji internetowej.
3. Aplikacja internetowa – właściwa aplikacja, z którą użytkownik wchodzi w interakcję. Zadaję użytkownikowi serię pytań otrzymanych od serwisu obsługi drzewa, na które może on odpowiedzieć tak / nie. Po wyczerpaniu pytań wyświetla wizerunek osoby, która pasuje do opisu wynikającego z udzielonych odpowiedzi.

6.1. Serwis bazodanowy REST API

Serwis bazodanowy zbudowany jest w oparciu o architekturę **REST** (ang. *Representional State Transfer*). Jest to popularne podejście do budowania interfejsów API i jest szeroko stosowane w architekturach mikroserwisów. REST jest zbudowany na protokole HTTP, który definiuje zestaw metod żądania, kodów odpowiedzi i nagłówków. Usługi RESTful wykorzystują te metody HTTP do wykonywania działań na zasobach identyfikowanych przez URI. Metody HTTP używane w REST to:

- GET: pobieranie zasobów
- POST: tworzenie nowych zasobów
- PUT: aktualizacja istniejących zasobów
- DELETE: usuwanie zasobów

Oprócz tych podstawowych metod HTTP, usługi RESTful wykorzystują również inne cechy HTTP, takie jak nagłówki, kody statusu i buforowanie, aby wdrożyć ograniczenia REST.

Jednym z kluczowych filarów REST jest użycie jednolitego interfejsu. Oznacza to, że zasoby są identyfikowane przez URI i dostępne przy użyciu standardowego zestawu metod. Zasoby są reprezentowane przy użyciu typów takich jak JSON lub XML. Pozwala to klientom zrozumieć dane zwracane przez serwer i umożliwia serwerowi zmianę reprezentacji

zasobu bez wpływu na klientów. Innym ważnym wymogiem REST jest bezstanowość. Oznacza to, że każde żądanie powinno zawierać wszystkie informacje potrzebne do wykonania żądania, bez polegania na jakimkolwiek stanie po stronie serwera. Pozwala to na wysoką skalowalność usługi, ponieważ żądania mogą być przetwarzane przez wiele serwerów bez konieczności dzielenia się stanem [12].

Stworzony na potrzeby aplikacji serwis posiada 3 tzw. endpointy (punkty końcowe), które obsługują żądania dotyczące 3 istniejących tabel w bazie:

- **/api/humans/**: obsługa żądań dotyczących tabeli *humans*. Pozwala na pozyskanie listy osób, etykiety osoby o podanym QID i danych osobowych na podstawie jej etykiety. Dodatkowo jest w stanie pozyskać link źródłowy do zdjęcia ukazujące wizerunek osoby o podanym QID, jeśli istnieje. Adres zdjęcia pozyskiwany jest przez wysłanie żądania do serwisu API Wikidata.
- **/api/proplabels/**: obsługa żądań dotyczących tabeli *propertylabels*. Pozwala na pobranie listy dostępnych atrybutów, listy atrybutów o danym typie, a także pojedynczego atrybutu na podstawie jego etykiety lub PID.
- **/api/entlabels/**: obsługa żądań dotyczących tabeli *propertyentitylabels*. Pozwala na pobranie listy dostępnych encji, a także pojedynczej encji na podstawie jej QID bądź etykiety.

Serwis zbudowany został w języku programowania Javascript w oparciu o framework sieciowy **Express.js** [13]. Jest to popularny framework aplikacji internetowych Node.js używany do tworzenia aplikacji i API po stronie serwera. Zapewnia prosty i elastyczny sposób budowania aplikacji internetowych, umożliwiając programistom tworzenie skalowalnych i wydajnych aplikacji. Głównym elementem Express.js jest system routingu, który umożliwia programistom definiowanie *handlerów* żądań HTTP lub punktów końcowych. Te *handlery* mogą być używane do obsługi różnych typów żądań, takich jak GET, POST, PUT, DELETE itp. Express.js zapewnia również funkcje pośrednie (ang. *middleware*), które mogą być używane do wykonywania różnych operacji przed lub po obsłudze żądania, takich jak parsowanie ciał żądań, obsługa plików cookie i sesji, logowanie, obsługa błędów itp.

6.2. Serwis obsługi drzewa decyzyjnego

Na potrzeby aplikacji internetowej stworzona została usługa, która na podstawie obecnego stanu aplikacji użytkownika oblicza następną optymalną wartość, generuje pytanie lub dokonuje predykcji, a następnie przesyła tą informację z powrotem do aplikacji. Serwis ten posiada 1 endpoint, który do poprawnej obsługi żądania potrzebuje obiektu z następującymi elementami:

- *Answered_yes*: Wartość logiczna definiująca, czy użytkownik odpowiedział na poprzednie pytanie twierdząca. Jeśli przyjmuje wartość negatywną, do serwis przy obliczaniu następnej optymalnej wartości będzie brał pod uwagę dalej opisany element *excluded*.
- *Excluded*: zawiera listę wartości, na które użytkownik w obecnej instancji odpowiedział negatywnie.
- *Instance*: zawiera obiekt, który reprezentuje obecne miejsce użytkownika w drzewie decyzyjnym. Kolejne pary atrybut – wartość są używane do znalezienia odpowiedniego poddrzewa, którego wartości będą sprawdzane pod kątem bycia najbardziej optymalnymi.

Po otrzymaniu żądania, serwis na podstawie otrzymanych danych generuje poddrzewo, z którego następnie oblicza najlepszą wartość atrybutu. Wartość wybierana jest na podstawie liczby „liści" znajdujących się pod tą wartością – wartość z największą liczbą liści zostanie wybrana jako następna. Jeżeli użytkownik wskaże, że ta osoba nie posiada danej cechy, wyeliminuje to największą ilość potencjalnych opcji, zawężając tym samym kryteria wyszukiwania. Jeśli wygenerowane poddrzewo będzie zawierało wyłącznie wartość tekstową, będzie to oznaczało, że trafiliśmy na wartość końcową – serwis jako odpowiedź zamiast pytania prześle etykietę osoby, która zostanie przetworzona przez aplikację internetową. W przypadku, gdy program natrafi na „ślepy zaułek" w drzewie (w wyniku wykluczenia wszystkich wartości w danym poddrzewie), serwis wyśle to aplikacji informację o nieznalezieniu osoby z danymi cechami.

Serwis został napisany w języku Python rozszerzonym o moduł **Flask** [14]. Flask jest frameworkiem sieciowym znanym ze swojego małego rozmiaru i lekkiej natury. Jest wyposażony w zestaw narzędzi i funkcji, które upraszczają proces budowania aplikacji internetowych w Pythonie. Flask oferuje programistom dużą elastyczność i jest doskonałym

wyborem dla tych, którzy są nowi w tworzeniu stron internetowych, ponieważ umożliwia im stworzenie aplikacji internetowej w jednym pliku Pythona.

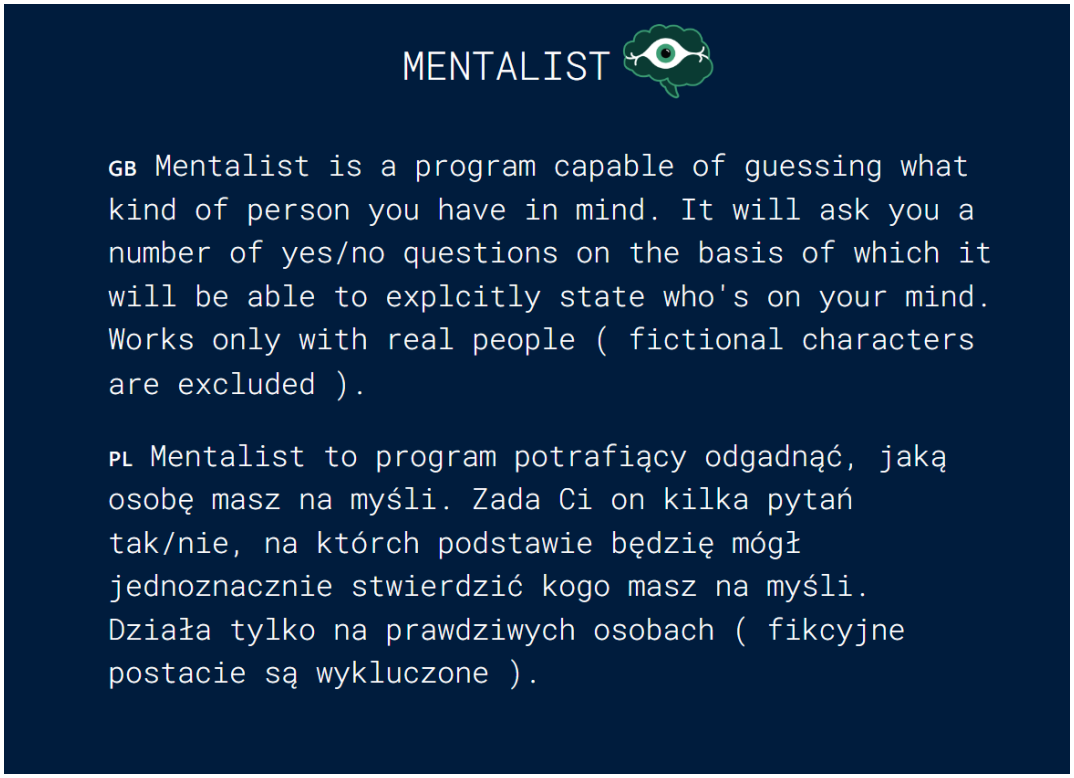
6.3. Aplikacja internetowa

Do stworzenia aplikacji internetowej wykorzystany został popularny framework języka Javascript o nazwie **React**. Pozwala programistom tworzyć komponenty UI wielokrotnego użytku i sprawnie zarządzać stanem tych komponentów [15]. React ułatwia tworzenie aplikacji internetowych poprzez deklaratywne i oparte na komponentach podejście do budowania UI, co prowadzi do ich szybszego rozwoju i łatwiejszego utrzymania. Dodatkowo, wirtualny DOM Reacta (ang. *Document Object Model* [16]) pozwala na wydajne aktualizacje UI, minimalizując ilość wymaganej rzeczywistej manipulacji DOM, tym samym poprawiając wydajność.

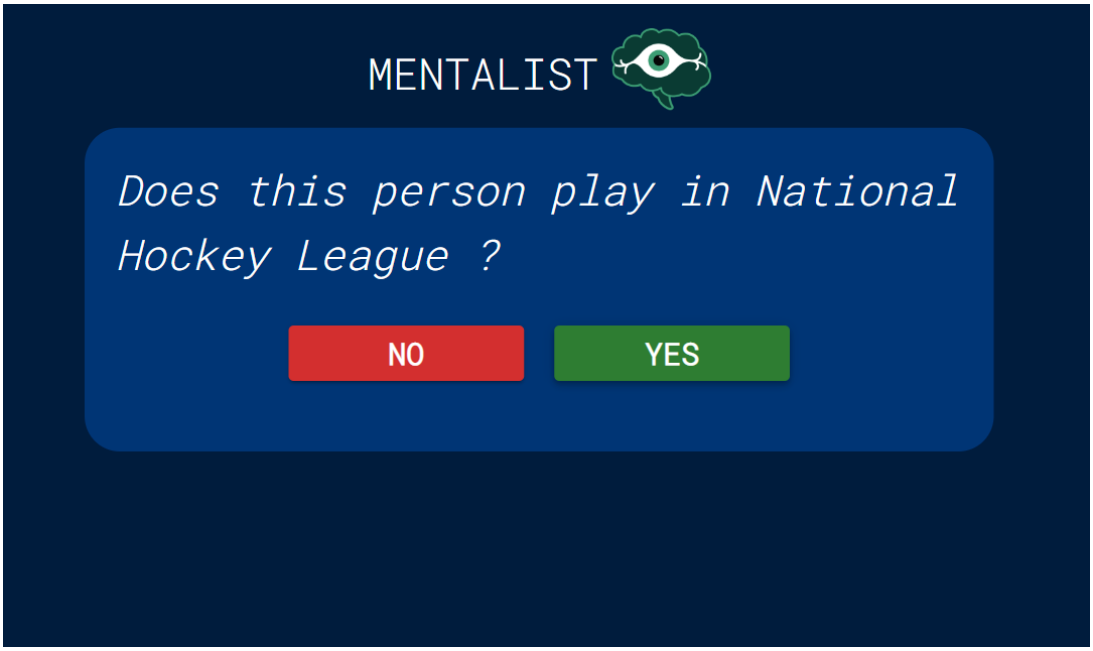
Aplikacja internetowa posiada 3 główne widoki:

- **Main**: strona główna aplikacji; pozwala na przejście do widoków **Game** i **About**.
- **About**: strona informująca użytkownika o przeznaczeniu aplikacji. Informacja zapisana jest w języku polskim i angielskim.
- **Game**: „właściwa” część aplikacji, w której użytkownik odpowiada na pytania postawione przez program w celu odgadnięcia osoby, o której myśli użytkownik

Rys. 5 Widok Main aplikacji internetowej



Rys. 6 Widok About aplikacji internetowej



Rys. 7 Widok Game aplikacji internetowej

Komponent **Game** przechowuje w swoim stanie informacje dotyczące odpowiedzi użytkownika i obecnego pytania przy użyciu tzw. *hooków* **useState** [17][18]. Użycie UseState deklaruje dwie wartości – zmienna reprezentującą stan i funkcję do jej zmiany. Użytkownik, wchodząc w interakcje z aplikacją, wywołuje funkcję zmieniającą stan zmiennej, a jej wartość jest natychmiastowo widoczna, jeżeli jest ona w jakimś stopniu użyta przy prezentacji strony. Oprócz UseState użyto również metody **UseEffect** [19], która wykonuje zawarty w sobie kod w momencie zmiany wskazanego elementu stanu aplikacji. W metodzie UseEffect zawarto proces pozyskiwania nowego pytania po podaniu odpowiedzi:

1. Stan formularza jest zmieniany na *false*, powodując zwinięcie formularza.
2. Wykonywane jest zapytanie do serwisu obsługującego drzewo decyzyjne, zawierające obecny stan gry.
3. Wartość pytania jest zmieniana na wartość otrzymaną w odpowiedzi.
4. Jeśli wartość pytania nie zawiera znaku „**?**”, program wysyła żądanie do serwisu REST API w celu pozyskania linku źródłowego do wizerunku osoby przewidywanej.
   4.1. Jeśli serwis API nie znajdzie linku źródłowego, lub jeśli odpowiedź serwisu obsługi drzewa nie będzie zawierała danych osoby (nie udało się znaleźć danej osoby), zostaje załadowany obrazek zastępczy. W przeciwnym wypadku zostaje załadowany wizerunek osoby.
   4.2. Po załadowaniu obrazu stan formularza jest zmieniany na *true*, rozwijając formularz zawierający nazwę osoby i jej wizerunek.
5. Jeśli wartość zawiera znak „**?**”, stan formularza jest zmieniany na *true*, rozwijając formularz z następnym pytaniem.

Stylizacja aplikacji została wykonana przy użyciu biblioteki stylizacyjnej **MaterialUI** [20]. Zapewnia ona szeroki zakres wstępnie zbudowanych komponentów React, które można wykorzystać do szybkiego stworzenia atrakcyjnego wizualnie i spójnego interfejsu użytkownika. MaterialUI zapewnia również system tematyczny, który pozwala programistom dostosować wygląd i sposób działania komponentów do brandingu aplikacji. Jedną z kluczowych zalet korzystania z MaterialUI jest łatwość użycia i spójny język projektowania, który zapewnia. Komponenty są dobrze udokumentowane i łatwe do wdrożenia, co pozwala programistom skupić się na budowaniu funkcjonalności aplikacji, a nie na spędzaniu czasu nad projektowaniem i układem. Dodatkowo, ponieważ komponenty są zaprojektowane zgodnie z wytycznymi Material Design [21], dając użytkownikowi spójne i intuicyjne doświadczenie w aplikacji.

7. Zakończenie

Celem niniejszej pracy licencjackiej było zaprezentowanie budowy i rozwoju aplikacji zdolnej do odgadnięcia osoby, o której myśli użytkownik, poprzez zadanie serii pytań tak/nie. Aplikacja została stworzona z wykorzystaniem algorytmów drzew decyzyjnych i architektury RESTful API. Wyniki uzyskane podczas testowania aplikacji na zbiorze danych osobowych były obiecujące - aplikacja była w stanie dotrzeć do osoby docelowej przy minimalnej liczbie pytań. W trakcie realizacji projektu napotkano kilka wyzwań, m.in. sposób wyboru atrybutów do podziału danych oraz optymalizacja działania programów wynikająca z rozmiarów zbioru danych. Przyszła praca nad tym projektem może obejmować włączenie bardziej zaawansowanych technik uczenia maszynowego, takich jak sieci neuronowe, w celu dalszej poprawy dokładności przewidywań aplikacji. Dodatkowo, aplikacja mogłaby zostać rozszerzona o bardziej wyrafinowane interfejsy użytkownika i zdolność do obsługi bardziej złożonych zbiorów danych. Projekt ten dostarczył cennych spostrzeżeń na temat zastosowania algorytmów uczenia maszynowego i architektury RESTful API w rozwoju rzeczywistych aplikacji. Jest nadzieja, że ta praca posłuży jako fundament dla przyszłych badań w tej dziedzinie.

8. Bibliografia

[1] https://www.wikidata.org/wiki/Wikidata:Main_Page [dostęp z dn. 13.04.2023r.]

[2] https://dumps.wikimedia.org/wikidatawiki/entities/ [dostęp z dn. 26.10.2022r.]

[3] https://www.mediawiki.org/wiki/Wikibase/DataModel [dostęp z dn. 14.04.2023 r.]

[4] https://qwikidata.readthedocs.io/en/stable [dostęp z dn. 14.04.2023r.]

[5] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, Database Systems: The Complete Book, Second Edition, Pearson, 2008, s. 1-12

[6] Simon Riggs, Gianni Ciolli, PostgreSQL 14 Administration Cookbook, Packt Publishing, 2022, s. 176-183

[7] https://www.npmjs.com/package/pg [dostęp z dn. 12.04.2023r.]

[8] https://www.npmjs.com/package/shelljs [dostęp z dn. 12.04.2023r.]

[9] Jiawei Han, Micheline Kamber, Jian Pei, Data Mining: Concepts and Techniques, Trzecia edycja, Morgan Kaufmann, 2011, s. 332-344

[10] https://pypi.org/project/psycopg2/ [dostęp z dn. 16.04.2023r.]

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom, Database Systems: The Complete Book, Second Edition, Pearson, 2008, s. 383-387

[12] Sam Newmann, Building Microservices. Second Edition, O'Reilly Media, 2021, s. 127-132

[13] https://expressjs.com/ [dostęp z dn. 15.04.2023r.]

[14] https://flask.palletsprojects.com/en/2.3.x/ [dostęp z dn. 15.04.2023r.]

[15] John Larsen, React Hooks in Action: With Suspense and Concurrent Mode, Manning, 2021, s. 1-10

[16] https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model [dostęp z dn. 15.04.2023r.]

[17] John Larsen, React Hooks in Action: With Suspense and Concurrent Mode, Manning, 2021, s. 13-17

[18] John Larsen, React Hooks in Action: With Suspense and Concurrent Mode, Manning, 2021, rozdział 2.

[19] John Larsen, React Hooks in Action: With Suspense and Concurrent Mode, Manning, 2021, rozdział 4.

[20] https://mui.com/material-ui/getting-started/overview/ [dostęp 15.04.2023r.]

[21] https://m3.material.io/get-started [dostęp 15.04.2023r.]

Spis listingów

Spis rysunków