Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Marosi Márk Dániel

2022

Szegedi Tudományegyetem

Informatikai Intézet

Domain specifikus szöveg feldolgozása kép alapú dokumentumokon

Diplomamunka

Készítette:
Marosi Márk Dániel
Gazdaságinformatika szakos hallgató

Témavezető:
Janurik Viktor Bálint
Tanszéki mérnök

Szeged
2022

Feladatkiírás

A digitalizáció és az automatizáció terjedésével egyre nagyobb az igény olyan programokra, melyek kép alapú dokumentumokról beolvasott szöveget képesek domain függően feldolgozni és osztályozni predikciók, szövegkörnyezet és a dokumentumon elfoglalt pozíció alapján.
A szakdolgozat célja egy ilyen program elkészítése egy tetszőlegesen választott domainnel.

Tartalmi összefoglaló

Tartalomjegyzék

1. fejezet

Bevezetés

A témaválaszatásomat az informatikai rendszerek elképesztő gyorsaságú fejlődésének gondolata alapozta meg. Egy olyan világban élünk, ahol az okos eszközök elkezdték kiváltani a manuális munkavégzési folyamatokat, vagy azokat egyszerűbbé tették. Minden nap a zsebünkben hordunk egy olyan kompakt eszközt, amely rendelkezik kamerával. Az okostelefonok kamerája, és egy erre fejlesztett applikáció együtt képes kiváltani egy hagyományos szkenner hardvert.

Ezen túlmenve, egyes felsőbb kategóriás telefonok már képesek fényképről felismerni szöveget, és opciót biztosítanak a szöveg kinyerésére is. Amennyiben előttünk van egy névjegykártya, és szeretnénk róla egy nevet vagy telefonszámot gyorsan kimásolni anélkül, hogy nekünk kelljen manuálisan begépelni, egyszerűen készítenünk kell róla egy fényképet, és amennyiben a telefon szöveget talál a képen, azt kimásolhatóvá és vágólapra illeszthetővé teszi a felhasználó számára, ezzel értékes időt spórova, továbbá a hibázás lehetőségét is csökkentve.

Mivel ezek a szoftverek egyre elterjedtebbek, szerettem volna mélyebben belelátni ebbe a témába, viszont egy nemrég történt tapasztalat csak felerősítette szakmai érdeklődésem a szövegfelismerés és feldolgozás kapcsán.

Egy határátkelésnél történt, hogy a személyi igazolványomat egy kis méterű, kompakt szkennelőgépbe helyezték, és kettő másodperc alatt minden adatom, ami a személyi igazolványomon volt, az a határrendészet szoftverébe került. Ez a folyamat egy olyan plusz lépést tartalmaz az előző, okostelefonos példához képest, hogy itt nem csak az igazolványon található szöveg került felismerésre és beolvasásra, mint egy nagy adathalmaz, hanem a szoftver képes volt ezt az adathalmazt megfelelően szétbontani és osztályozni, felismerte hogy melyik adat a név, melyik adat az állampolgráság, és így tovább.

Szakdolgozatom célja, hogy ezt a témát körbejárjam, felkutassam a legújabb technológiákat és egy ilyen folyamatot be tudjak mutatni egy tetszőlegesen kiválasztott domainnel.

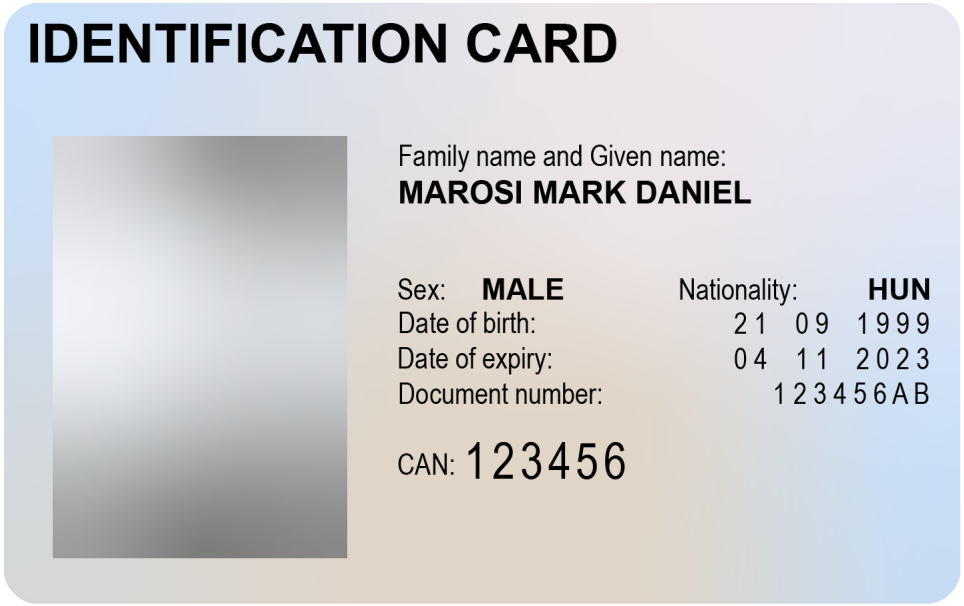
Ennek megfelelően szakdolgozatomat öt fő részre tagoltam. A dolgozat első részében a jelenleg legismertebb és legmodernebb technológiák utáni kutatásom eredményét fogom részletezni, mélyebben kifejteni azt, hogyan működnek azok a hasonló felhasználási célú redszerek, melyeket jelenleg használnak. A második részben rávilágítok a jelenleg bárki számára elérhető szövegfelismerő rendszerek hiányosságaira, továbbá bemutatom és összehasonlítom azokat a technológiákat, keretrendszereket, amelyeket érdemes lehet felhasználni a projektemben. Harmadik lépésben a projektem megvalósításának részleteit, komponenseit, felépítését fogom bemutatni, valamint megindoklom a domain választásom. Végül lépésről lépésre bemutatom a program működését egy konkrét esetre, valamint a program futásának elvárt eredményét.

2. fejezet

Jelenlegi rendszerek

2.1. Jelenlegi rendszerek bemutatása

Egy átlagos felhasználónak kép alapú szövegbeolvasásra jelenleg telefonos vagy web alapú applikációk segítségével van lehetősége. Ez a legegyszerűbb módszer, mert könnyen kezelhető, nem igényel hozzáértést, ingyenesen igénybe vehető, gyors és megbízható. A következőkben egy általam készített, nem valós adatokat tartalmazó minta igazolványon fogom tesztelni az egyik applikációt.



2.1. ábra. minta igazolvány

A weboldalon található egy fájl feltöltésre alkalmas mező, ide kell tallózni a képet, melyből szeretnénk kinyerni a szöveget, ezután pedig el kell indítani a beolvasási folyamatot, mely körülbelül 5 másodpercet vesz igénybe. A folyamat végén egy szövegmezőből kimásolhatóvá válik a képről kinyert szöveg. Az általam feltöltött képről kinyert szöveg:

> IDENTIFICATION CARD Family name and Given name: MAROSI MARK DANIEL Sex: MALE Nationality: Date of birth: Date of expiry: Document number: CAN: 123456 HUN 21 09 1999 04 11 2023 123456AB

Az eredményt megvizsgálva megállapítható, hogy a képből kinyert szöveg hibátlan, a betű és a szám alapú adatok is helyesen jelennek meg. Viszont azt is észrevehetjük, hogy az adatok rendezetlenek, nem minden adatról azonosítható be, hogy pontosan mit jelent.

2.2. Jelenlegi rendszerek hiányosságai

Tegyük fel, hogy egy olyan szituációban találjuk magunkat, ahol több személyi igazolványt kell digitalizálnunk, például egy Excel táblázatban eltárolni a kártyán olvasható adatokat. Erre három esetet fogok felvázolni. Az első a klasszikus módszer, ahol közvetlenül a személyi igazolványról írjuk át az adatokat a táblázatba, kézzel begépelve. A második módszer az előbbiekben bemutatott webapplikáció segítségével történne, a harmadik eset pedig egy olyan program használatát feltételezi, melyet a szakdolgozatom keretében fogok elkészíteni.

Amennyiben az első, klasszikus módszert választjuk, kézzel kell beírnunk minden egyes adatot, betűről betűre, számról számra. A három módszer közül ez a leghosszabb és legtöbb emberi hibalehetőséget rejtő módszer. Hibázhatunk az adatok olvasásánál, hibázhatunk az adatok begépelése során, valamint ott is hibázhatunk, hogy nem jó cellába visszük fel az értékeket, elcsúszunk valahol.

A második módszer az első módszerben felvázolt három hibalehetőségből kettőt minimálisra csökkent, ezek a beolvasási és begépelési hibák. A beolvasást egy olyan technológiára alapozva végzi a szoftver, amit a szakdolgozat következő fejezeteiben fogok részletesebben bemutatni. Fontos megjegyezni, hogy egyes weboldalak lehetőséget biztosítanak egyszerre akár több kép feltöltésére, ezzel is gyorsítva a folyamatot. A technológia rendkívül pontos, emiatt eltekinthetünk attól, hogy beolvasási hiba történjen, azaz eleve rossz adat kerüljön a táblázatba. A második, begépelési hibalehetőséget kiküszöböli az, hogy a szoftver a beolvasott szöveget másolhatóvá teszi, így szimplán a másolás és beillesztés műveletek segítségével vihetjük fel az adatokat a kívánt cellába. Tehát amennyiben a szövegfelismerés hibátlan volt, úgy minimálisra csökken annak az esélye is, hogy a másolás során elrontsunk valamit. Az egyetlen fennálló hiba továbbra is az, hogy a másolt adatot rossz cellába illesztjük be, összekeverünk két, látszólag hasonló alakiságú adatot, például a születési időt a kártya lejárati dátumával.

A harmadik módszer mind a három hibalehetőségre megoldást biztosítana. A program futása alatt végrehajtott folyamat első része teljes mértékben ugyan úgy történne, mint a második módszer esetében: a feltöltött képről vagy képekről kinyeri a szöveget ugyan azon a technológián alapulva. A program ezután nem a nyers adathalmazt adja vissza a felhasználónak, mint a második esetben, hanem tovább dolgozik azon, és a futás eredménye egy olyan csv vagy excel kiterjesztésű fájl lenne, ahol minden adatkategória (név, állampolgárság, stb.) egy oszlop lenne, és az oszlopnév alatt helyezkedne el a hozzá tartozó adat, például az állampolgárság oszlopban a HUN szöveg. Ezzel teljesen megszűnne minden emberi hibalehetőség, hiszen a teljes folyamatot elvégezné helyettünk a szoftver. Ennek a módszernek további pozitív hozadéka, hogy az adatokat képes az elvárásainknak megfelelően formázni, például a dátumot, amely 30 06 1979 alakot vesz fel a kiolvasás után 1979.06.30 vagy egyéb tetszőleges formára tudná hozni, tovább csökkentve a manuális teendőket.

3. fejezet

Az OCR technológia

3.1. Az OCR működése

Az OCR, azaz Optical Character Recognition (magyarul Optikai Karakterfelismerés) egy olyan technológia, amely képes bármilyen képről vagy digitális dokumentumról az írott vagy nyomtatott szöveget felismerni és kinyerni. Az OCR egyik legnagyobb haszna, hogy képes kiváltani a kézi adatbevitelt, jelentős időt spórolva és megszüntetve az emberi hiba lehetőségét is.

Képzeljünk el egy könyvet, amit egy hagyományos szkennelővel beszkennelünk. A folyamat eredménye digitális képek sorozata lesz, melyeket könnyedén tudunk digitális eszközeink között mozgatni, vagy akár nyomtatóval sokszorosítani tudjuk, de szerkesztni, szöveget keresni vagy kimásolni belőle nem tudnánk. Ha egy olyan szkennerünk lenne, amelyben OCR alapú szövegfelismerés is lenne, akkor a könyvet könnyedén digitalizálhatnánk és nem képeket kapnánk eredményül, hanem egy olyan szöveges dokumentumot, amely teljesen szerkeszthető és kereshető.

Az Optikai Karakterfelismerés folyamata több lépésből tevődik össze. Az első lépésben egy osztályozási folyamat történik, ahol a kiválasztott képet vizsgálva a világos területeket háttérnek, a sötét területeket pedig szövegnek minősíti. Ebből kifolyólag az OCR pontosságát tovább javíthatjuk azzal, ha már a karakterfelismerés előtt az inputként választott képet fekete-fehérré alakítjuk.

Második lépésben a szövegként minősített területeken egy olyan keresés indul, amelynek célja az alfabetikus karakterek (betűk) és numerikus karakterek (számok) azonosítása, ez történet karakterről karakterre, vagy szavanként (karakter láncokként) is. Az azonosítás egyik leggyakoribb módszere a mintaillesztésre alapul. Ennél a módszernél egy olyan adathalmazból dolgozik az algoritmus, amely sok különböző betűtípus- és szövegképmintát tárol egy adatbázisban úgy, mint egy sablont. Mikor a képen alakzatot próbál felismerni, összehasonlítást végez a tárolt sablonok alakzatával, és a legnagyobb egyezést

mutató karakternek fogja minősíteni a képen látható alakzatot. Ez a módszer akkor igazán pontos, ha az input egy olyan kép, melyen ismert betűtípusokkal jelenik meg gépelt, nyomtatott szöveg, mivel a betűtípusok és kézírási stílusok száma végtelen, és lehetetlen minden típust az adatbázisban rögzíteni. Amennyiben kézzel írt szöveget adunk bemenetként, akkor a pontos eredmény eléréséhez egy olyan algoritmusra van szükség, amely figyelembe veszi a karakterek jellemzőit is. Ilyen jellemzők például a betű írására használt vonalak, azok irányai, elhelyezkedései, metszéspontjai, görbületei és hurkai. Ezen tulajdonságokat az algoritmus minden felismerhető karakterről tárolja, majd a keresett alakzatot is felbontja ugyan ezekre, és megkeresi a tárolt karakterek közül azt, amellyel a legtöbb jellemző megegyezik. Ezt a folyamatot Intelligent Character Recognition (ICR), magyarul Intelligens Karakterfelismerés névvel illetik.

Utolsó lépésben a dokumentum teljes szerekezeti képének függvényében a felismert karakterek önmagukban, szavakba, mondatokba vagy szövegblokkokba rendezve kerülnek tárolásra.

3.2. Az OCR pontosságának mérése

Az előzőekben bemutattam, hogyan képes az OCR egy szöveget tartalmazó képet gépi szöveggé alakítani, de felmerülhet bennünk a kérdés, hogy mégis mennyire pontos az eredmény, amit kapunk egy ilyen konverzió során. A karakterfelismerés csupán képpontról képpontra vizsgálja a képet, és a betűk alakjából vonja le a végső következtetést, arra viszont nem képes, hogy a dokumentum teljes kontextusát felismerve megállapítsa, hogy a szöveg, amit kinyert, az pontosan mit is jelent, és helyesnek bizonyul-e az adott környezetben. Emiatt az OCR gyakran hibázhat, és ezek a hibák pont a szövegfelismerés által adott előnyöket csökkentik.

Legegyszerűbben úgy határozható meg a pontosság, hogy az OCR kimeneti eredményét összehasonlítjuk a képen szereplő szöveggel. Tegyük fel, hogy a képen szereplő szöveg 100 karakterből áll. Amennyiben az OCR által adott eredményben mind a 100 karakter egyezik az eredeti szövegben szereplővel, akkor azt mondhatjuk, hogy az OCR pontossága 100%. Amennyiben 99 karaktert sikerült eltalálnia az szövegfelismerőnek, úgy a pontosság 99%. Tehát egyszerű arányosítással is kiszámolható egyfajta pontosság.

Most bemutatom a két leggyakrabban használt metrikát, melyek erre a logikára épülnek.

3.2.1. CER - Character Error Rate (Karakter hibaarány)

A CER mutató azon karakterszintű műveletek minimális számát mutatja meg, amelyek szükségesek a bemeneti szöveg hibátlan kimenetté való konvertálásához. A CER számításához használt képlet:

$$CER = \frac{T}{T + C} * 100$$

Ahol T az OCR eredményéből érkező karakterek a bemenettel azonos karakterekre való transzformációk számát jelöli (tehát ezek olyan karakterek, melyek helytelenül lettek felismerve), C pedig a helyesen felismert karakterek száma.

Példa:

Felismerendő szöveg: abcdefg-123
OCR kimenet: abcdef9-1Z3
Mivel a g betűt 9-es számkarakternek állapította meg, továbbá a 2-es számot Z betűnek, így 2 transzformációra lesz szükségünk, tehát T=2.
A helyesen felismert karakterek száma 9 (a,b,c,d,e,f,-,1,3), ezért C=9.

$$CER = \frac{2}{2 + 9} * 100 = 18.18$$

Ebben a példában 18%-os értéket vesz fel a CER mutató, természetesen ez a szám minél kisebb, annál jobb.

3.2.2. WER – Word Error Rate (Szóhibaarány)

Hasonlóan a CER-hez, ennél a metrikánál azt vesszük figyelembe, hogy hány szó szintű műveletre van szükség ahhoz, hogy az OCR folyamat eredménye teljesen megegyezzen a bemeneti szöveggel. Bár a WER érték a szavak metrikáját méri, nem a betűkét, de ha belátjuk, hogy ugyan azon betűk sorozatából kapjuk a szavakat, akkor jogosan feltételezhetjük, hogy a WER és a CER metrikák jól korrelálnak egymással.

A WER mérésére ugyan azt a képletet használjuk, mint a CER érték méréséhez, de a T paraméter a helyes szóra történő transzformációk számát, a C paraméter pedig nem a helyes karakereket, hanem a teljes terjedelmében helyesen felismert szavak számát jelöli.

3.2.3. További metrikák az OCR pontosságának megállapításához:

– SER - Symbol Error Rate (Szimbólum hibaarány):

  • Ez a metrika kifejezetten azt vizsgálja, hogy a szövegben szereplő szimbólumok, különböző írásjelek milyen arányban kerültek helyesen felismerésre.

– Text-Based F1 Score (Szövegalapú F1-pontszám):

  • Ez a mérőszám a felismert szöveg helyes részarányának, illetve a helyesen felismert bemeneti szöveg részarányának a harmonikus átlagát számolja.

– Keystroke Saving (Billentyűleütés megtakarítás):

  • Azt méri, hogy ha egy kézi bevitelen alapuló folyamatot egy OCR alapú rendszerrel váltunk ki, akkor hány billentyűleütést spórolunk meg.

Fontos megjegyezni, hogy az előbbiekben bemutatott metrikák nem adnak minden esetben valós képet a különböző OCR modellek működéséről, hiszen a beolvasott dokumentumok minősége, valamint a tény, hogy kézírást vagy nyomtatott szöveget adunk bemenetként mind erősen befolyásoló tényezők az OCR kimenetének helyességében.

4. fejezet

Feladat ellenőrző, kiértékelő és naplózó modul

4.1. Egy szálon futó proceduriális megoldás

A fejlesztés első lépése az alapos kutatómunka volt, azonban ez minden fázisában jelen volt. Kezdetben a komponensek közötti kommunikáció adatbázison keresztül történt, a kód procedurálisan futott és kezelte az adatbázis lekérdezéseket. A validáló kód egy kezdetleges részlete:

```
def validate_commands(db):
    db = db_connect()
    command = get_history(db)
    correct_commands = get_answers(db)

    if str(command).find(str(correct_commands)) != -1:
        print("the correct command ran")
    else:
        print("the commands are not correct")
```

Ekkor még adatbázis alapú kommunikációra akartuk építeni a projektet, és a kezdetleges kód bővült a feladatokhoz tartozó kiadandó parancsokkal, illetve a felhasználó által kiadott parancsok listájának lekérdezését végző metódussal, ami az aktuális munkamenet kezdetétől lekéri a teljes parancselőzményt:

```
def get_history(db):
    current_user = get_user_data()[0]
    start_time = get_user_data()[1]
```

```
current_time = datetime.datetime.utcnow().\
               strftime('%Y-%m-%d %H:%M:%S')

get_history_cursor = db.cursor()
get_history_cursor.execute(
    "SELECT parancs FROM query " +
    "WHERE time_stamp >= \"" + str(start_time)
    + "\" " +
    "AND time_stamp <= \"" + str(current_time)
    + "\" " +
    "AND user_id=" + str(current_user) + ";"
)

result = get_history_cursor.fetchall()

tuples_joined_in_array = [''.\
        join(tups) for tups in result]
history_string = r"{}".format(
    str(tuples_joined_in_array).replace("'", "").\
replace(", ", "").\
replace("[", "").\
replace("]", "")

return history_string
```

Funkcionalitás szempontjából ez tekinthető az első működő béta verziónak, amely képes volt a felhasználó által kiadott parancsok kiértékelésére. Miután az előzőek megvalósításra kerültek, a sok adatbázis lekérdezés és a folyamatos insert-ek miatt egy performancia teszt során kiderült, hogy egy diszkrét idő után, a karakterekénti beillesztések miatt az adatbázis rekordszámai nagy iramban fognak nőni, és az adatbázis kezelhetetlenné, a kiadott parancsok pedig olvashatatlanná fognak válni.

Ennek hatására kezdtünk el másik megoldás után kutatni, szóba került, hogy Message Queue-t alkalmazzuk az eszközök közötti kommunikációra. Ennek nagy előnye, hogy a kommunikáció az MQTT protokoll szabványa szerint (ISO/IEC 20922:2016)[1] gyorsabb, mint adatbázison keresztül. Szavazat után a fejlesztőcsapat a Rabbit Message Queue alkalmazása mellett döntött, ez azt eredményezte a továbbiakban, hogy az eddigi kódbázis

[1] a protokoll olyan TCP/IP vagy más hálózati protokollon fut, amely képes rendezett, veszteségmentes, kétirányú kapcsolatot biztositani.

nagy része funkcionalitását vesztette, és alapjaiban kellett újjáépíteni a teljes modult.

4.2. Több szálon futó objektumorientált megoldás

A modulok közötti kommunikáció módjának megváltoztatása után hamar kiderült, hogy az ellenőrző modul bonyolultabb lesz, mint amit egy procedurális megvalósítás engedne. Éppen ezért a kezdetekben két osztály került bevezetésre, a Validator, ami később User-re lett átnevezve, és a MetaCommunication osztályok, melyek feladata nevükből is felismerhetően, rendre a feladatok ellenőrzése és a weboldallal való kommunikáció volt.

```
class Validator (threading.Thread):
    def __init__(self, threadID, name):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name

    def run(self):
        print("Starting " + self.name)
        init(self.threadID)


class MetaCommunication(threading.Thread):
    current_user = None
    selected_task = None
    correct_command_list = r''
    com_ports_in_use = []

    def __init__(self):
        threading.Thread.__init__(self)
        self.threadID = "web-python"
        self.name = "web-python"

    def set_correct_command_list(self):
        db = db_connect()
        get_correct_commands_cursor = db.cursor()
        get_correct_commands_cursor.execute(
            "SELECT command FROM correct_commands " +
            "WHERE lesson_id=\"" + str(lesson_id) + "\";"
```

```
        )
        self.correct_command_list = r"{}".format(
            get_correct_commands_cursor.fetchone()[0])
        db_close(db)

    def run(self):
        print("Starting " + self.name)
        web_python_communication_listener(self)
```

A MessageQueue használata koncepcionálisan egyszerű, a kommunikációban két típusú ágenst különböztetünk meg, egy úgynevezett "publisher" vagy "content creator" és egy "subscriber" vagy "consumer" résztvevőt. A felosztás magától értetődő volt, a web-es frontend a publisher, a konzolos backend, a validator és a logoló modulok mind consumerek. A MetaCommunication osztály a message queue-ba csatlakozó metódusa:

```
def init(queue):
connection = pika.BlockingConnection(
    pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='test')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)

channel.basic_consume(queue=queue,
                      on_message_callback=callback,
                      auto_ack=True)

print(' [*] Waiting for messages.
       To exit press CTRL+C')
channel.start_consuming()
```

Azonban komplikációk léptek fel az MQ implementálása során. Habár a csatorna figyelésére aszinkron kapcsolatot használ, a programszálat a start_consuming() metódus aktívan fogja, így az első csatornára való kapcsolatnyitás teljesen megállítja a programot. Ennek kiküszöbölésére az MQ-kat több szálon kezdtem kezelni, a különböző COM

portokon való kommunikációt egy-egy külön szálon figyeltem, amihez egy újabb osztály bevezetésére volt szükség a QueueListener osztályra.

További problémák akadtak a szálak megszüntetésekor is. A jelenleg belépett felhasználó adatai jelen vannak a példányosított objektumokban, ezért a kijelentkezés után ezeket meg kell szüntetni, hogy helyükre újakat lehessen létrehozni a megfelelő felhasználói adatokkal. Ezek az objektumok a Thread osztály leszármazottjai, belépési pontjukban pedig a RabbitMQ csatornái felé nyitott start_consuming() metódussal figyelnek, ami nem szakítható meg a Thread osztály join() metódusával. Ennek következtében először a stop_consuming() metódussal le kell zárni a kapcsolatot, majd ez után lehet a szálat is zárni.

```
class QueueListener(threading.Thread):
#is_waiting_for_join = 0
def __init__(self, com_port, parentObject):
    super().__init__()
    self.name = "queue_listener_thread_on" +
                str(com_port)
    self.com_port = com_port
    self.parentObject = parentObject

def run(self):
    web_device_communication_listener(
        self.parentObject,
        self.com_port)
```

Ezt az osztályt a Validator osztály példányosítja annyi példányban, ahány COM port csatlakoztatva van a rendszerhez. (Ez jelenleg hardcode-oltan 6 COM portot jelent 1-6 között) Továbbra is akadály volt az, hogy ezek a threadek – a BlockingConnection miatt – nem reagáltak a Thread osztály join() metódusára. A probléma orvosolására egy, a problémát megkerülő megoldásra volt szükség. Végül egy igen egyszerű és az infrastruktúrára kis kihatással lévő megoldás született, a Validator osztály (majd később a QueueListener osztály) egy új adattagot kapott is_waiting_for_join névvel, amelynek érteke kezdetben 0, a felhasználó kilepésekor érteke 1-re módosul. A szál konkrét lezárását a következő logika végzi.

```
if user_object.is_waiting_for_join == 1:
channel.close()
```

Amely a QueueListener, a Message Queue-ra figyelő metódusának, callback függvényében helyezkedik el. Ez azonban még nem teljes értékű megoldás, hiszen, ha a fel-

használó kilépett a rendszerből, a következő üzenetet már a következő felhasználó fogja küldeni. Ez azt eredményezi, hogy amíg a rendszer üres járatban van, a program 6 felesleges szálat futtat folyamatosan, továbbá a régi felhasználó szálainak lezárása és az új felhasználóhoz tartozó szálak indítása JIT módon történne. Így a felhasználó kilépését feldolgozó kódrészlet kiegészült egy blokkal, amely behirdet egy szóköz karaktert a Message Queue-kra:

```
if str(body.decode('UTF-8').split(',')[0]) == "logout":
    joining_thread = thread_list.get("Thread-" + str(
                    body.decode('UTF-8').split(',')[1]))
    print("DEBUG thread object: " + str(joining_thread))
    joining_thread.is_waiting_for_join = 1

    for com_port in joining_thread.com_ports_in_use:
        closing_connection = pika.BlockingConnection(
            pika.ConnectionParameters('localhost'))
        closing_channel = connection.channel()
        closing_channel.queue_declare(queue=com_port)
        closing_channel.basic_publish(exchange='',
                                      routing_key=com_port,
                                      body=b" ")
        closing_connection.close()
```

4.3. Az ellenőrzés és kiértékelés folyamata

A kiértékelési folyamat egy felhasználó bejelentkezésekor indul. A bejelentkezési eseményt a frontend úgy jelzi a modul számára, hogy a következő sémára alapuló "login" üzenetet küld:

```
login,user\_id,lesson\_id
```

A user_id a felhasználó egyedi azonosítója, a lesson_id pedig a megfelelő Cisco tananyaghoz tartozó labor feladat. A feladat neve a Cisco modul nevéből – azaz Introduction to Networks (ITN), Connecting Networks (CN), Scaling Networks (SCAN) és Routing and Switching Essentials (RASE) – és a modulon belüli fejezet sorszámából áll, ezáltal a feladathoz tarozó tananyagrész és vice versa könnyen megtalálható. Amennyiben a felhasználó nem kíván meglévő feladatot elvégezni, az utolsó paraméter nem kerül elküldésre, ekkor a kiértékelő modul nem indul el.

A feladat neve alapján még a példányosítás során, a konstruktor becsatlakozik az adatbázisba, ahonnan kiolvassa a feladathoz tartozó parancsokat, amiket a felhasználónak ki kell adnia, hogy az általa választott feladatot teljes egészében elvégezze. A lesson_id elsődleges kulcsa a helyes parancssorozatot tartalmazó correct_commands táblának. A tábla többi oszlopa a COM portokat reprezentálják, ezek értéke a kiadandó parancsok. Miután a lekérdezés megtörtént, a kiválasztott feladathoz tartozó sort a program eltárolja egy szótárban, amelynek kulcsai a COM portok, az értékei a parancsok. Ez a szótár a User osztály adattagja. Ezzel egyidőben inicializál két változót, amik a jelenlegi megoldottságot, és egy százalékos inkremenst tárolnak, amit egy helyes parancs kiadásakor kell hozzáadni a teljes százalékösszeghez.

Az osztálypéldány készen áll, hogy Thread-ként elinduljon. Ha a felhasználó a rendelkezésre álló feladatok közül választott, akkor a QueueListener és Logger osztály konstruktora is meghívásra kerül, ha önálló feladatot old meg, csak a Logger osztály kerül meghívásra. A parancsokat kiértékelő metódus a QueueListener osztálypéldányok belépési pontjában kerül meghívásra, tehát akár a COM portok szempontjából szimultán feladatkiértékelésre is képes lenne. Fontos kérdés volt, hogy a karakterenkénti üzeneteket hogyan lesz képes a kiértékelő modul kezelni. A megoldást nyujtó feltételvizsgálat két tényen alapul, a sorvége karakter két vezérlőkarakterből áll ("\n" és "\r") és az üres sort – tehát enter ütést – nem akarjuk kiértékelni.

```
listener.lines_of_code += (body.decode('UTF-8'))
  if len(listener.lines_of_code) > 2:
    if listener.lines_of_code[
       len(listener.lines_of_code) - 2:] == "\n\r":
      validate_commands(listener, user_object, com_port)
```

A felhasználó által beírt parancsot a lines_of_code változó tárolja, és erre meghívja a parancsot kiérékelő metódust.

4.1. ábra. Néhány kiértékelés kimenete konzolon

A függvény először megvizsgálja, hogy a kiadott parancs tartalmaz-e rövidítést, amit egy JSON fájlból olvas be egy globális szótárba még a program indulásakor. A szótár kulcsai a rövidített parancsok lesznek, értékei a rövidítések teljes alakos változata. Ha talál rövidítést a parancsban, azt kicseréli a kulcs-érték pár alapján, majd tovább halad az ellenőrzéssel.

```
if listener.lines_of_code == "conf t":
  listener.lines_of_code = "configure terminal"
else:
  # Split the command, to get the first
  split_command = listener.lines_of_code.split(sep=" ")
  print("split command: {}".format(split_command))
  # If the command given is a shorthand known to us
  if split_command[0] in abbreviations:
    print("its an abbreviation")
    # Change the shorthand command to the full command
    replace_command = abbreviations['{}'.\
                      format(split_command)]
    print("replace_command: {}".format(replace_command))
    listener.lines_of_code.replace(split_command[0],
                                   replace_command, 1)
    print("final: {}".format(listener.lines_of_code))
```

4.2. ábra. Egy rövidített parancs cseréje

Végigjárja az adatbázisból kiolvasott parancsokat és ellenőrzi, hogy megtalálható-e benne a megfelelő alakra hozott parancs.
Ha megtalálható, ezt jelzi a felhasználó számára, és a User osztályban létrehozott százalékmérőt növeli a megfelelő inkremenssel.
Ha nem található meg, üzenetet küld a felhasználó számára a frontendre. Ez azonban nem egy hibaüzenetet, hiszen lehetséges, hogy állapotlekérdező parancsot futtatott, hogy egy beállítást ellenőrizzen az eszközön.
A lines_of_code változó értékét üríti, és egy végső feltételvizsgálatot végez. Ellenőrzi, hogy az a parancs, amivel meghívták, 100%-osan megoldottra növelte-e a feladatot. Ha

igen, üzen a felhasználónak a feladat elvégzéséről. Az is_waiting_for_join változót 1-re állítja, a message queue-khoz nyitott csatornára ezáltal meghívódik a close() pika névtérbeli metódust. Az objektumpéldány belépési metódusában folyamatosan futó metódus erre a változóra írt feltételvizsgálat miatt kilép és az azt futtató programszál is.

4.4. Logoló modul

A rendszer fontos része a logoló modul, aminek segítségével a felhasználók által a különböző COM portokra kiadott parancsok naplózásra kerülnek. Az üzeneteket a modul külön message queue-n kapja meg, ami fontos abból a szempontból, hogy ha egy queue-t két consumer olvas, akkor az egyikőjüktől érkező acknowledge hatására a másik consumer az adott üzenetet nem kapja meg. Ebben az esetben előfordulhat, hogy a hálózati eszközök hamarabb olvassák az üzenetet, és ez nem jut el a logoló modulhoz, ergó nem kerül naplózásra a kiadott parancs, avagy vice versa, a parancs naplózásra kerül, de az eszközön nem kerül futtatásra.

4.3. ábra. Logolás az adatbázisban

A naplózás mechanikájában egy puffer került bevezetésre, ami biztosítja, hogy abban az esetben, ha gyorsabban érkeznek az üzenetek, mint ahogy az adatbázisba való insert lefutna, azok nem vesznek el. A pufferből csak akkor kerül törlésre az üzenet, ha az sikeresen az adatbázisba lett írva:

```
[...]
for item in command_buffer:
    if log_to_db(item, db, user_id, com_port):
        command_buffer.pop(command_buffer.index(item))
[...]

def log_to_db(command, db, user_id, com_port):
    insert_cursor = db.cursor()

    sql = "INSERT INTO command_history(user_id, command, com)
           VALUES (%s, %s, %s)"
```

```
    val = (user_id, command, com_port)

    print('DEBUG_LOOGER command: ' + command)
    insert_cursor.execute(sql, val)

    try:
        db.commit()
    except mysql.connector.Error as err:
        print(err)
        return False
    return True
```

4.5. Összefoglalás

A szakdolgozatban leírt projekt lényege, hogy segítséget tudjunk nyújtani a hallgatóknak akár a Cisco hálózatépítő kurzusban, vagy akár a Cisco ipari vizsgára való készülésben. A fejlesztés a szakdolgozaton túl is folytatódik, célunk, hogy egy olyan rendszer álljon a hallgatók rendelkezésére, amivel produktívan lehet tanulni, kísérletezni.

A fejlesztés során lehetőségem nyílt nem csak csapatban, de új technológiákkal is dolgoznom. Köztük a RabbitMQ, aminek használata számomra újdonság és kihívás volt és a Python objektumorientált használata, amire ezelőtt még nem volt szükségem, lehetőségem. Illetve, eddigi tudásomat is lehetőségem nyílt kamatoztatni, a git verziókezelés, a gitlab-runner és a hozzá tarozó CI/CD leírók használata, a docker és docker-compose, és a hyper-v virtualizáció terén.

5. fejezet

Adatbázis sémák

5.1. Adatbázis sémák

A program a következő adatbázisokkal dolgozik:

- lab_tasks
- correct_commands
- running_configs
- startup_configs

A lab_tasks tábla tartalmazza az elérhető feladatok címét, a Cisco tananyag fejezet azonosítóját, amivel a hozzá tartozó tananyag könnyen megtalálható, és COM[1-6] oszlopokat, amelyeknek az értéke a feladat szerinti eszköz hostneve.

A correct_commands tábla azokat a parancsokat tartalmazza, amelyeket a különböző COM portokon ki kell adni, a feladat helyes elvégzéséhez.

A running_configs tábla azokat a "show running config" parancs kimeneteket tartalmazza, amelyeket a helyesen konfigurált eszközöknek produkálnia kell.

A startup_configs tábla pedig a hibaelhárítási feladatokhoz tartozó helytelen beállításokat előállító parancsokat tartalmazza, amelyeket a feladatmegoldás előtt a program futtat az eszközön.

5.1. ábra. Séma relációs ábrázolása

6. fejezet

Az alkalmazás működése

6.1. Az alkalmazás működése

A program belépési pontja a main.py, ami indulásakor példányosítja a MetaCommunication osztályt. Ez a példány becsatlakozik a RabbitMQ-ba, ahol egy message queue-n figyel, és vár egy felhasználó csatlakozására. A csatlakozást követően készen áll a használatra.

Egy felhasználó csatlakozásakor a MetaCommunication osztály user_connection() metódusa meghívódik, ezzel példányosítva a User osztályt. A User osztály konstruktora a szükséges adattagokat beállítja (felhaszáló azonosítója, kiválasztott feladat (ha van), helyes parancsok listája, százalékos inkremens).

6.1. ábra. A program indulása előtti és utáni message queue-k

A példányosítás után a vezérlés visszatér a MetaCommunication osztály user_connection() metódusához, ami a Thread típusú User példányt elindítja. Ezzel a User objektum run metódusa meghívásra kerül, ami a programszálat elindítja. Annyi felhasználóhoz tartozó Logger osztálypéldányt készít, ahány COM porton keresztül kommunikál az eszközökkel, továbbá ugyanennyi QueueListener osztálypéldány is létrejön, ha a felhasználó választott feladatot. Azaz, a jelenlegi felállás szerint egy felhsználóhoz – a hat COM porthoz – kétszer hat programszál tartozik, így a MetaCommunication és fő szállal együtt tizennégy vagy nyolc szálon fut a program.

6.2. ábra. A programszálak vizualizálva
(Készült a PyCharm Concurrent Activities Diagram segítségével)

A QueueListener példányok a hozzájuk tartozó message queue-kat figyelik, és az azon keresztül érkező parancsokat fogadják. Feldolgozásra meghívják rájuk a validate_commands() metódust, ami kiértékeli azt.

Amikor a felhasználó elvégezte a feladatot, a szintjelző 100%-ra vált, és üzenetet küld a felhasználónak erről. Ekkor a message queue-kat figyelő szálak feleslegessé válnak, ezért ezeket egyesítjük az őt példányosító szálakkal. Ez úgy érhető el, hogy a QueueListener példány is_waiting_for_join tulajdonságát 1-re állítjuk, ami megállítja a futtatását.

A szemétgyűjtést a Python beépített automatikus szemétgyűjtője végzi, ami referenciaszám alapján azokat az objektumokat, amikre már nem hivatkozik semmi, törli a memóriából. A dokumentációban leírtak alapján a szemétgyűjtés explicit meghívásra kerül, ha az újonnan létrehozott objektumok száma meghaladja a már jelenleg memóriában lévők 25%-át. Képletesen felírva:

$$current\_objects \times 0.25 < new\_objects \Rightarrow garbage\_collection \tag{6.1}$$

Ez a százalékos határ a program esetében ideális. A program indulásakor létrejön a MetaCommunication egy példánya. Egy felhasználó bejelentkezésekor kétszer hat objektum jön létre (hat QueueListener és hat Logger a hat COM porthoz). Amikor a felhasználó kijelentkezik, a következő felhasználó bejelentkezéséig két eset állhat fenn.

1. Az automatikus szemétgyűjtés nem fut le, tehát a következő felhasználó bejelentkezésekor az előző felhasználóhoz létrehozott objektumok még a memóriában vanank.

2. Az automatikus szemétgyűjtés lefutott és az előző felhasználó objektumai a következő bejelentkezésekor már nincsenek a memóriában.

Az első eset az, ami vizsgálatot igényel: Egy felhasználó aktív kapcsolata során 1 + 2 x 6 = 13 objektum van a memóriában, ezek pedig nem törlődtek. A következő felhasználó bejelentkezésekor 2 x 6 objektum keletkezik. A képletet alkalmazva:

$$13 \times 0.25 < 12$$
$$3.25 < 12$$

Az egyenlőtlenség igaz, tehát a szemétgyűjtés legkésőbb a következő felhasználó bejelentkezésekor meghívódik.

7. fejezet

Továbbfejlesztési lehetőségek

7.1. Továbbfejlesztési lehetőségek

A szakdolgozat keretein belül implementálásra nem került funkcionalitások:

- A jelenlegi kód a hálózati erőforrásokhoz (MySQL adatbázis szerver, RabbitMQ) hardcode-oltan csatlakozik, ezért a kód átírása nélkül más fizikiai infrastruktúrára nem lehet telepíteni.
- Az adatbázisba leképezett CCNA kurzushoz kapcsolódó laborfeladatok száma igen alacsony, mivel a Word dokumentumokból és PDF fájlokból a konfigurációs lépések kiemelése nem szkriptelhető, ehhez nagy időbefektetés szükséges.
- Parancsok kiadása az eszközökön a modul által:
  - A felhasználó kijelentkezése után, az eszközök konfigurációjának törlése és az eszközök újraindítása az "indulási konfiguráció" (startup configuration) betöltésével.
  - A felhasználó által meghívható teljes rendszer visszaállítás arra az esetre, ha valamelyik felhasználó által kiadott parancs elérhetetlenné teszi a rendszert (pl.: jelszó beállításakor félreütött karakter miatti kizárás az eszközről).
- A projektben megírt osztályok és metódusaik, mind egy forrásfájlban találhatóak, amely nehezíti a kód olvasását. Ennek több forráskódra bontása, osztályok mentén, ezek csomagokba rendezése és névterek használata, nagyban javítaná a projekt átláthatóságát.

– Az idő előrehaladtával a Cisco tananyag verziózásának követése (a jelenlegi rendszer a 6-os verzión alapul). Ez megvalósulhat akár több tábla használatával, vagy akár több adatbázis séma létrehozásával. Ennek megvalósítása biztosítaná, hogy a modul a jövőben ne váljon elavulttá.

Irodalomjegyzék

[1] Cisco Networking Academy,

v6 *Connecting Networks, Introduction to Networks, Routing and Switching Essentials & Scaling Networks* Instruktori forrásfájlok

[2] Python projekt struktúrálás és PEP

https://docs.python-guide.org/writing/structure/

[3] MySQL 8.0 Reference, SQL parancs szintaxisok,

https://dev.mysql.com/doc/refman/8.0/en/

[4] MySQL Cursor Objects,

execute és fetch metódusok használata és paramétereik https://mysqlclient.readthedocs.io/user_guide.html

[5] Docker Dokumentáció, parancsok szintaxisa és paramétereik,

https://docs.docker.com/

[6] Python 3 threading – Thread-based parallelism,

https://docs.python.org/3/library/threading.html

[7] Python 3 Classes,

https://docs.python.org/3.9/tutorial/classes.html

[8] MQTT protokoll

https://mqtt.org/

[9] RabbitMQ Tutorials,

Work Queues, Publish/Subscribe https://www.rabbitmq.com/getstarted.html

[10] Pika AMPQ protokoll implementációs dokumentáció

https://pika.readthedocs.io/en/stable/

Nyilatkozat

Alulírott Kersmájer István szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztési Tanszékén készítettem, gazdaságinformatika diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat / diplomamunkámat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Szeged, 2023. január 30. .....

aláíras

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani Janurik Viktor témavezetőmnek, aki a fejlesztés minden lépésében hasznos tanácsokkal látott el. A segítségéért a tesztkörnyezet összeállításában, a sok konzultációért és szakmai tanácsaiért, amelyek nélkül ez a projekt nem jutott volna tovább az "initial commit"-on.

Köszönöm továbbá két fejlesztőtársamnak, Orbán Veronikának és Csóti Zoltánnak, akikkel a fejlesztés produktívan tudott haladni, és kódváltozásaimhoz igazították saját munkájukat. A hallgatóknak, akik használni fogják az új rendszert, visszajelzéseikért az esetleges problémákról, és további fejlesztési ötleteikért.