Mesterségesintelligencia-kutatások a Miskolci Egyetemen

Elemző és robotizált folyamatautomatizálási rendszer fejlesztése nagy terhelésű ügyfélszolgálatok számára

Kutatási jelentések
2022/1

Miskolci Egyetemi Kiadó
2022

Mesterségesintelligencia-kutatások
a Miskolci Egyetemen

Szerkesztette:
Prof. dr. Kovács László

Mesterségesintelligencia-kutatások a Miskolci Egyetemen

Elemző és robotizált folyamatautomatizálási rendszer fejlesztése nagy terhelésű ügyfélszolgálatok számára

Kutatási jelentések
2022/1

Szerkesztette: Prof. dr. Kovács László

Miskolci Egyetemi Kiadó
2022

ISBN 978-963-358-262-6

TARTALOMJEGYZÉK

Előszó

Jelen kiadvány a 2020-1.1.2-PIACI-KFI-2020-00165 projekt keretében a Miskolci Egyetem Gépészmérnöki és Informatikai Karának Informatikai és Matematikai Intézeteiben a 2021. július – 2022. április közötti időszakban elvégzett kutatások eredményeit foglalja össze.

Az adminisztratív és informatikai folyamatoknál a hatékonyságnövelés egyik fő eszköze a folyamatautomatizálás minél szélesebb körben történő megvalósítása. A robotizált folyamatautomatizálás (RPA, Robotic Process Automation) célja az üzleti folyamatok magas szintű automatizálása szoftverrobotok (botok) létrehozásával és futtatásával. Az ilyen robotok a nagy volumenű, ismétlődő feladatokat nagyobb pontossággal és hatékonysággal végzik el, mint az ember. Az RPA-eszközöket a rutinfeladatok automatizálására tervezték, amelyek strukturált adatokat tartalmaznak és amelyeknek determinisztikus eredményei vannak, a végrehajtandó lépéseket pedig szabályok vezérlik. Emellett fontos jellemzőjük, hogy az RPA-eszközök képesek felhasználói interakciókat végrehajtani az ezzel járó adatkezelési műveletekkel együtt. A robotizált folyamatbányászat (RPM, Robotic Process Mining) célja az automatizálható folyamatok feltárása gépi tanulási eszközökkel. A folyamatbányászó módszerek kifejlesztése egy egyelőre nagyrészt még felderítetlen problémakör. Az egyik legígéretesebb megközelítési módszer a tevékenységnaplók alapján történő eseményfeltárás.

A projekten dolgozó kutatócsapatnak több főbb feladatkört is le kellett fednie. Egyrészt meg kellett vizsgálnia, hogy az egyes eseménynapló-modell formátumok milyen mértékben alkalmasak az igényelt információk kiemelésére. A második részproblémakör a rugalmas napló-előfeldolgozó keretrendszer kidolgozása volt, mely a különböző formátumú napló állományokat egy egységes objektum szerkezetre hozza. A harmadik lépés az eseménygráf-sémák feltárásának klasszikus, automataalapú módszerének a kidolgozása volt. Ennek során új funkciók kerültek be az alap eljárásba, jelentősen növelve annak hatékonyságát. Mivel a kutatás egy további célja a neurális háló alapú sémafeltárás elemzése volt, itt elsőként a klaszszikus eseménysort előrejelző neurális hálók értékelését végeztük el. Az eredmények alapján az MLP- és LSTM-hálók kerültek kiválasztásra a további vizsgálatokhoz. A kutatás egy további fázisában olyan új neurálisháló-architektúra és feldolgozási modell került kidolgozásra, amely már alkalmas az XOR és AND típusú sémaelágazások felderítésére is. A projekt keretében emellett, a következő időszak kutatásait előkészítendő, elemzéseket végeztünk a GAN-hálótípus alkalmazhatóságára és a magyar nyelvű szöveget tartalmazó dokumentumok NLP-alapú

témakör azonosítására vonatkozólag is. A projekt keretében kidolgozott mintarendszereket Python nyelven fejlesztettük ki.

A projekt lehetőséget adott egy hatékonyan együttműködő csapat kiépítésére is, amely több különböző szakterület képviselőit fogja össze. A csoport tagjai, a kötet szerzői:

Dr. Baksáné dr. Varga Erika, kutatási terület: adatelemzés, adatbányászat
Csépányi-Fürjes László, kutatási terület: számítógépes nyelvészet
Dr. Kovács László, kutatási terület: adatbányászat, szemantikai modellek
Dr. Mileff Péter, kutatási terület: szoftverfejlesztés, adatelemzés
Dr. Radeleczki Sándor, kutatási terület: automaták, logika
Dr. Samad Dadvandipour, kutatási terület: adatelemzés, adatbányászat

A különböző szakterületek képviselőinek együttműködése számos érdekes megközelítést bontakoztatott ki, remélhetőleg az olvasóinknak is sok hasznos ötlettel fog szolgálni az elkészült összefoglaló.

Miskolc, 2022. május 17.

Prof. dr. Kovács László
szakmai projektvezető

FELHASZNÁLÓI TEVÉKENYSÉGEK MONITOROZÁSA

MILEFF PÉTER

Az ERPA projekt által megfogalmazott egyik fontos cél, hogy a rendelkezésre álló, valamilyen információs rendszerből kinyert aktivitásnaplók alapján bizonyos események előre megjósolhatók legyenek valamilyen hatékonyan konfigurálható mesterséges neurális hálózatot alkalmazó modell segítségével. A mesterséges intelligencián alapuló informatikai rendszerek egyik kulcskérdése a megfelelő adathalmaz, hiszen ez teszi lehetővé a ráépülő folyamatok, a tanuló algoritmusok tervezését, a megfelelő betanulást. A magas minőségű adathalmaz tehát kulcsfontosságú, ezért kiemelt figyelemmel kell megközelíteni a problémát.

Jelen projekt során az adatokat az üzemeltetett rendszerben elvégzett felhasználói tevékenységek rögzítése, monitorozása fogja nyújtani. A rögzített adatok, a munkafolyamat-lépések egy magasabb szintű szekvenciába, tranzakcióba szerveződnek. Mivel a rendszert egyszerre több felhasználó is használja, így egy olyan naplófájl fog létrejönni, amelyekben véletlenszerűen keverednek a különböző tranzakciókhoz tartozó tevékenységek.

Jelen dokumentum áttekinti a felhasználói aktivitás naplózásának legfontosabb kulcskérdéseit, problémáit és egyaránt gyakorlati oldalról is bemutatja a naplófájlok egy lehetséges feldolgozásának kérdéseit.

1. A kutatás célja és lépései

Napjaink modern világában a vállalatok ma már komplex informatikai rendszereket használnak az ügyviteli folyamataik támogatására. Nem ritka az olyan vállalkozás, ahol akár több mint 100 ember végzi a napi tevékenységeket. Általános jellemzője ezeknek a rendszereknek, hogy több tucat különböző típusú folyamatot végeznek az alkalmazottak. Idővel a vállalat fejlődése a folyamatok komplexitásának növekedését eredményezi. Csökken az átláthatóság, az érthetőség és idővel sokszor észrevehetetlen szűk keresztmetszetek alakulhatnak ki. Ilyen komplex rendszerek irányítása egyre több problémát vet fel mind biztonságtechnikailag, mint pedig a folyamatok követhetősége szempontjából. Az utóbbi idők eredményeképpen létrejött felhasználói aktivitásfigyelés (User Activity Monitoring – UAM) és az RPA-megoldások ezekre a problémákra próbálnak megoldást találni. A felhasználóiaktivitás-figyelés (UAM) megoldások olyan szoftvereszközök alkalmazását jelentik, amelyek figyelemmel kísérik és nyomon követik a végfelhasználói viselkedést az eszközökön, hálózatokon és más, a vállalat tulajdonában lévő informatikai erőforrásokon.

A kutatási munka célja, hogy részletesen áttekintse a felhasználói aktivitás szerepét, megmutassa azokat a normákat, amik rendelkezésre állnak és egy olyan irányt javasoljon, amely megfelelő eredményt jelenthet a gyakorlati alkalmazás során is.

A tervezett kutatási célok és lépések a következők:

- UAM-rendszerek főbb csoportjainak áttekintése.
- A felhasználóitevékenység-felügyelet működésének általános bemutatása: mit jelent az, hogy naplózás. Milyen főbb megközelítéseket szokás alkalmazni.
- A felhasználóitevékenység-felügyelet legfontosabb eszközeinek feltárása és rövid ismertetése.
- Célszerű megvizsgálni az aktivitásfigyelés gyakorlati megközelítését. Magát a naplókat, az eseményelőzményeket biztosító rendszereket, valamint az olyan rendszereket, amelyekhez nekünk kell összeállítani az előzményeket.
- További fontos terület a gyakorlati megvalósításhoz kapcsolódó problémák és akadályok feltérképezése és áttekintése.
- Aktivitási naplók hiánya és a naplók hiányosságai.
- Az esetazonosítóval kapcsolatban felmerülő problémák.
- Az aktivitással kapcsolatos problémák.
- Az időbélyeggel kapcsolatos problémák.
- Naplók ismertebb hibái: hiányzó aktivitások, hiányzó időbélyeg, befejezetlen folyamatok stb.
- A probléma neurális hálózattal való megoldásának rövid ismertetése, figyelembe véve a tanítóminta teljességét és az erőforrásszükséglet

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

Az UAM-rendszereket célszerű két csoportba sorolni az alkalmazási céloknak megfelelően:

Felhasználói monitorozás információmegőrzés, biztonság, rosszindulatú visszaélések megakadályozása érdekében:

Számos szervezet a felhasználóiaktivitás-figyelő eszközöket alkalmazza a bennfentes fenyegetések észlelésének és megakadályozásának, akár akaratlanul, akár rosszindulatú szándékkal. A monitorozás és az alkalmazott módszerek köre a vállalat célkitűzéseitől függ. Az aktivitáskövetés megvalósításával a szervezetek hatékonyabban követhetik nyomon a munkavállaló online tevékenységeit, biztosíthatják az erőforrások megfelelő felhasználását, és megállíthatnak minden

olyan gyanús tevékenységet, amely a termelékenység csökkenéséhez vezet, és veszélyezteti a szervezetet. Könnyebben azonosíthatják a gyanús magatartást és mérsékelhetik a kockázatokat, mielőtt azok adatvédelmi jogsértést eredményeznének, vagy legalább időben a károk minimalizálása érdekében.

Felhasználói aktivitások monitorozása RPA-folyamatok támogatására:

Ebben az esetben az információ gyűjtésének célja egy olyan információs adathalmaz létrehozása, amely alapja lehet egy későbbi RPA- vagy Process Mining elemzésnek, gyakorlati megvalósításának. Legfőbb jellemzője az információk pontos és jól strukturált tárolása. A kinyert adathalmaz lehetőséget nyújt a szűk keresztmetszetek gráfelméleti módszerekkel való feltárására, a folyamatok automatikus reprodukálására, mesterséges intelligenciával való támogatására, továbbá esetleges bekövetkező események előrejelzésére. A továbbiakban a projekt keretein belül ezzel a témacsoporttal foglalkozunk mélyebben.

2.1.1. Felhasználóitevékenység-felügyelet működése

A felhasználói aktivitások figyelésére általában valamilyen altivitásfigyelő szoftvert alkalmazunk. Az aktivitásfigyelő szoftver rögzíti az alkalmazások és programok használatát a felügyelt munkaállomáson. A képernyőn megjelenő felhasználói tevékenységek egy előre kidolgozott és jól strukturált naplóba kerülnek. A naplók tehát információs adatbázisok, amelyek minden olyan tevékenységet tárolnak, amelyek aznap történtek. A mai modern technológiának köszönhetően számos lehetőség, megoldás áll rendelkezésre a monitorozásra, a tevékenységek figyelemmel kísérésére és kezelésére. Néhány fontosabb technika:

- **Naplózás és elemzés:** a felhasználóiaktivitás-figyelés leginkább elterjedt formája. Ebben az esetben egy klasszikus szöveges fájlban, esetleg adatbázisban tároljuk a történt eseményeket. Minden esemény és folyamat jól definiált. A készülő naplófájl jól strukturált formában tárolja az információkat a későbbi feldolgozhatóság és analizálhatóság érdekében. Talán azt mondhatjuk, hogy több más alkalmazott technika alapjául szolgál.
- **A munkamenetekről készült videófelvételek:** a felhasználói aktivitásról videó stream formájában felvétel készül. Fontos cél a későbbi elemezhetőség, az esetleges biztonsági visszaélések megelőzése, valamint az elvégzett interakciók esetleges reprodukciója.

- **Képernyőkép rögzítése:** hasonló megfontolások alkotják a technika célját, mint a videóalapú megközelítésnél. Ennél a módszernél képeket rögzítünk, amely alapot ad a további elemezhetőségre. Több gyakorlati megvalósításban a rögzített képernyőképeken az UAM-eszközképek az adott aktivitás egy dobozzal való bekeretezésére. Például a felhasználó által kiválasztott elem, nyomott gombok stb.
- **Billentyűzethasználat naplózása:** a lenyomott gombok naplózása olyan esetekben lehet fontos, amikor a felhasználók által használt vállalati vagy egyéb rendszer kevesebb grafikus interakciót vár el, több folyamat esetleg kézi adatbevitelt / gépelést vár el.
- **Hálózati csomagellenőrzés:** talán inkább az UAM-rendszerek biztonsági csoportjába tartozó megoldás. A felhasználók, a rendszer használói által küldött és fogadott csomagok naplózása. A forgalom ellenőrzése számos adatvédelmi lehetőséget nyújt a vállalat számára, megvédhet akár az esetleges visszaélésektől is.
- **Kernel monitoring:** az operációs rendszer kerneljének alacsony szintű monitorozása. Alkalmazása főként szintén biztonsági kérdésekben indokolt.

Az összes összegyűjtött információt a vállalati irányelvek és a felhasználói szerepkör határain belül kell megvizsgálni, hogy meghatározhatók legyenek az összefüggő folyamatok, a későbbiekben elemezni kívánt tevékenységek, valamint az esetleges nem kívánt felhasználói tevékenységek. Az, hogy mit jelent valójában a „nem megfelelő felhasználói tevékenység", az mindig az adott vállalattól függ. A vállalatnak kell kiválasztani azokat az UAM ellenőrzési megoldásokat, amelyek bármire kiterjedhetnek a személyes webhelyek felkeresésétől vagy a munkaidőben történő vásárlástól a kényes vállalati adatok, például szellemi tulajdon vagy pénzügyi információk lopásáig.

2.1.1.1. Felhasználóitevékenység-felügyelet eszközei

Számos eszköz használható a felhasználói aktivitás figyelemmel kísérésére vagy támogatására. Ezek az eszközök az általános biztonsági szoftveralkalmazásoktól a munkamenetek és tevékenységek nyomon követésére szolgáló célzott eszközökig terjednek, és minden felhasználó számára teljes naplózási nyomvonalat hoznak létre. Vannak privilegizált fiókbiztonsági megoldásokként is ismert eszközök, amelyek célja a privilegizált fióktevékenységek figyelemmel kísérése és biztosítása, valamint a házirendek kezelésének központosítása.

A legjobb felhasználóiaktivitás-figyelő eszközök közé tartoznak a *valós idejű* figyelőrendszerek. Ezek az eszközök valós időben (gyakorlatilag azonnal) naplózzák a felhasználói tevékenységeket a háttérben, figyelik a felhasználói képernyőt a háttérben, és amikor célzott tevékenységre kerül sor, figyelmezteti az adminisztrátort vagy a menedzsert, vagy elvégzik a megfelelő tevékenységeket. Ezek lehetnek egyszerű, például az operátort támogató javaslatok, esetleg valamilyen szinten automatizált háttérfolyamatok elvégzése. Más esetekben pedig akár gyanús tevékenységek riportálása az informatikai és biztonsági csoportoknak. A valósidejűség nélkül a kockázatok bizonyos esetekben észrevétlenek maradhatnak, miközben a felelős informatikai csoport más ismert kérdésekkel foglalkozik. A mai technológiának köszönhetően nem szükséges, hogy egész informatikai csapat foglalkozzon az aktivitások monitorozásával és elemzésével. Egy jó UAM-megoldás kidolgozása, amely támogatja a felhasználói aktivitás figyelemmel kísérését, elemzését, riportálását és esetleges beavatkozását, nagymértékben egyszerűsíti a problémát.

2.2. Aktivitásfigyelés gyakorlati megközelítése

Bármely olyan tevékenységnek, amely a későbbiekben a felhasználói aktivitás eredményére szeretne építeni, legfontosabb kiindulópontja minden esetben az úgynevezett eseménynapló lesz. A leginkább gyakorlatban is alkalmazott technika az úgynevezett *Process Mining*, amelynek célja az adatok folyamatszempontú elemzése. Olyan kérdésekre keresi a választ, mint például a „A folyamat jelenleg milyen állapotban van?", „Vannak-e felesleges lépések, amelyeket ki lehetne küszöbölni?", „Hol vannak a szűk keresztmetszetek?" És „Vannak-e eltérések a lefektetett és előírt folyamatszabályoktól?"

Annak érdekében, hogy az eseménynaplóban lévő adatokat bármilyen formában is elemezni lehessen, azokat valamilyen folyamatspecifikus megközelítéssel kell a logba írni. Ez lehetővé fogja tenni, hogy a későbbiekben az eseménynaplóban szereplő lépésről el lehessen dönteni, melyik folyamathoz tartozik és azon belül pedig az adott folyamat melyik lépése volt.

Talán a témakör egyik legfontosabb kérdése lehet az, hogy *„Honnan érkeznek az adatok az eseménynaplóba?"*

A kérdés válasza rögtön két csoportra bontja az információforrást és az aktivitásfigyelés módszerét.

2.2.1. Naplókat, eseményelőzményeket biztosító rendszerek

Olyan rendszerek, amelyek már önmagukban tartalmazzák az eseménynaplózást, melyek sok esetben akár nagyon jól konfigurálhatók is. Például az ügyfélkapcsolat (CRM), az IT-szolgáltatáskezelés (ITSM), a rendeléskezelés és a munkafolyamat-rendszerek általában ebbe a kategóriába tartoznak. Az aktivitásfigyelés és folyamatbányászat szempontjából ezek a rendszerek a megvalósítási spektrum könnyebb oldalához tartoznak, hiszen az adatok nagy valószínűséggel megfelelő minőségben már rendelkezésre állnak naplók, adattáblák (előzménytáblák) és egyéb megoldások formájában, azok sokszor közvetlenül felhasználhatók. Például egy ITSM-rendszerben minden esetben van jegyszám, továbbá minden állapotváltozást rögzítenek.
Sok esetben ezek az adatok könnyen exportálhatók CSV-fájlként és közvetlenül importálhatók a folyamatbányász eszközbe előzetes feldolgozás nélkül.

2.2.2. Rendszerek, amelyekhez nekünk kell összeállítani az előzményeket

Itt, a spektrum nehezebb végén olyan adatbázis-központú rendszereket találunk, mint az Enterprise Resource Planning (ERP) és bizonyos egyedi vagy legacy típusú rendszerek. Ezekben az esettörténetek nem állnak rendelkezésre feldolgozáskész formában, mert nem így lettek tervezve és kialakítva már a kezdetektől fogva. Ezért minden esetben valamilyen egyedi megoldást kell készíteni arra vonatkozóan, hogy a releváns adatokat az üzleti adatbázis-táblákban megtaláljuk, és létrehozzunk egy saját eseménynaplót az elemzéshez.

Ezen rendszerek további kihívása, ha a folyamatot több informatikai rendszer is támogatja, használja. Tipikus példa lehet, amikor a vásárlási igényeket az ERP-rendszerben kezeljük, a számlákat pedig külön, pénzügyi rendszeren keresztül támogatjuk. Ha két (vagy több) rendszer adatait kívánjuk egyesíteni, akkor a legfontosabb dolog, amire figyelni kell, hogy miként tudjuk követni az esetet ezeken a különböző rendszereken. Például, ha az ERP-rendszer minden esetben vásárlási rendelési számot használ, a pénzügyi rendszer pedig minden esetben számlaszámot használ, akkor a számlában találni kell egy olyan hivatkozást, amelyre a megrendelés vonatkozik.

A megoldást ezen rendszerek esetében két útra lehet bontani:

- **A rendszer forráskódjának kiegészítése:** egyik legjobb megoldás lehet, ha rendelkezünk a rendszer forráskódjával és a folyamatelemzés szempontjából igényelt és kritikus pontokra eseménynaplózást helyezünk el a kódban. Mivel a teljes forráskód módosítható, így az események naplózása maradéktalanul

megoldható. A módszer meglehetősen nagy hátrányát jelenti, hogy egy jól működő rendszer forráskódjába ismételten belenyúlni sokszor veszélyes és a rendszer teljes újratesztelése időigényes lehet. Ez függ attól, hogy a korábbi fejlesztőkörnyezet hogyan lett kialakítva, esetleg kidolgozásra került-e valamilyen automatikus tesztelési eljárás, stb. Tovább nehezítheti a fejlesztést az is, hogy sok esetben bár rendelkezésre áll a forráskód, de a rendszer fejlesztése óta számos év telt el, így a forráskód és az egyes támogató függvénykönyvtárak frissítése elengedhetetlen. Végül a módosított rendszert (ismét) üzembe kell helyezni.

- **Felhasználóiaktivitás-figyelő szoftver alkalmazása:** egy másik alternatívát képvisel az, ha létező aktivitásfigyelő szoftvert alkalmazunk. A megoldás előnye, hogy nem kell a működő rendszer kódját vagy bármely részét módosítani, hanem a felügyelni kívánt rendszerekre, állomásokra csak az UAM-szoftvert kell telepíteni és konfigurálni. A mai modern UAM-szoftverek a felhasználói interakció már számos fajtáját képesek kezelni/naplózni többékevésbé. Sajnos mégsem mondhatjuk ki, hogy a probléma maradéktalanul megoldásra került, a következő okok miatt: bár a felhasználók interakciói naplózásra kerülnek, az események folyamatlépésekhez való kötése a legtöbb esetben nem tud maradéktalanul megvalósulni. Ennek oka, hogy a megtörtént esemény – legyen az egy megnyomott gomb – olyan egyedi azonosítóval kell legyen ellátva, amely köthető a folyamathoz, mint annak egy lépése. Mivel a felhasználói interfész elemeinek azonosítói a fejlesztéskor kerültek meghatározásra, nem biztos, hogy úgy lettek kialakítva, hogy azok alkalmasak legyenek a későbbi naplózáshoz.

A gyakorlatban a fent vázoltak miatt leginkább valamilyen UAM-szoftvert próbálnak alkalmazni a megoldás megvalósításához azért, mert a forráskód nem áll rendelkezésre, esetleg más cég készítette a szoftvert. Fontos szót ejteni arról, hogy annak megállapítása, hogy milyen UAM-szoftvert alkalmazunk, függ a naplózni kívánt szoftver megvalósításától, amelyek lehetnek vastag, illetve vékony kliensalapúak. Míg a korábbi években az informatikai szoftverek jellemzően vastag kliens módszerekkel lettek fejlesztve, ma már leginkább vékony kliens, web-alapú megoldásokkal találkozunk. A két megközelítés eltérő UAM-megoldást kíván meg. Vastag kliensek esetében alig található a piacon bármilyen UAM szoftver. Ebben az esetben a szoftver erősen függ az operációs rendszertől, amely nagymértékben megnehezíti azt, hogy egy külső szoftver naplózást készítsen a kívánt másik szoftverről. Sok esetben a képernyőképmentések jelentik a megoldást, de ez is számos további problémát vethet (felbontás, UI-azonosítók hiány stb.) fel.

Szerencsére a világ vékony kliens irányba való elmozdulása új lehetőségeket nyitott meg a naplózás területén is. Számos szoftvercsomag, beépülő modul áll rendelkezésre a kívánt eredmény elérésére.

2.3. A megvalósításhoz kapcsolódó problémák és akadályok

Egy jól megvalósított RPA rendszer általában komplex megvalósítási folyamatot igényel. Számos részfeladat, amelyek önmagukban is bonyolultak lehetnek, természetüknél fogva bizonytalanságokat, illetve problémákat rejtenek magukban. A következőkben néhány ilyen jellegű fontosabb problémát emelünk ki.

2.3.1. Aktivitási naplók hiánya

Mint ahogyan az korábban említésre került, a megfelelő adathalmaz a mesterséges intelligencián alapuló informatikai rendszerek egyik kulcskérdése, hiszen ez teszi lehetővé a a tanuló algoritmusok tervezését, a megfelelő betanulást.

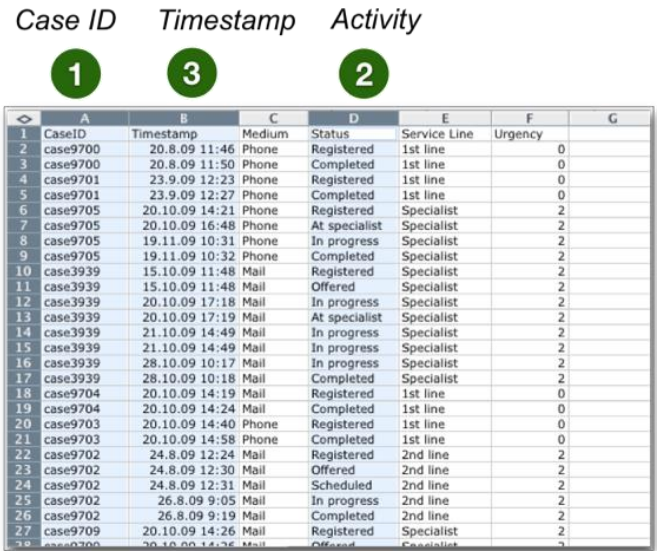
Az egyik legfontosabb kérdés talán az, hogy lesz-e megfelelő minőségű adathalmaz a projekt során? A kérdés jogos aggályt fogalmaz meg, hiszen egy már évekkel ezelőtt üzembe állított rendszer vélhetően nem úgy lett tervezve, hogy a felhasználói aktivitások egyszerűen kinyerhetők belőle. Ráadásul a több komponensből álló rendszer különböző komponensei nem ugyanúgy viselkednek, eltérő felépítéssel rendelkeznek, melyek a naplózási aktivitásban is minden bizonnyal eltérnek. Természetes követelmény, hogy a projekt során a partner rendszereiben megtörténjen a megfelelő részletezettségű aktivitásnaplók készítésének beépítése, azonban mivel ez önmagában véve is időigényes feladat, nem várható, hogy már a kezdetben is rendelkezésre álljanak használható aktivitásnaplók.

A mintarendszert ezért olyan módon kell megtervezni, hogy képes legyen a valósághoz közelítő, vagy azzal megegyező adatokat generálni. Ezáltal a kezdeti adathalmaz hiánya feloldható, a kutatási feladatok, az algoritmusok tervezése nem blokkolódik.

2.3.2. Aktivitási naplók hiányosságai

Egy aktivitásnapló alapvetően három kötelező elemből kell álljon: esetazonosító (case id), aktivitás (activity) és egy időbélyeg (timestamp). Minden ***eset*** egy folyamat végrehajtási példányának egy lépését jelenti. Például egy rendelési folyamatban, egy megrendelés kezelése egy esetet jelent. Nagyon fontos kritérium, hogy minden esetnél tudni kell azt, hogy melyik az a folyamat, amihez globálisan tartozik. Az ***aktivitás*** a folyamat egyik lépése. Például egy dokumentumkészítési folyamat a következő lépésekből állhat: „Létrehozás", „Frissítés", „Küldés",

„Jóváhagyás", „Átdolgozás kérése", „Felülvizsgálat", „Közzététel", „Elvetés" (különböző személyek, mint pl. szerzők és szerkesztők). Ezen lépések némelyike egy esetben többször is megtörténhet, de nem kell mindegyiknek minden alkalommal megtörténnie. A folyamat során végrehajtott különböző folyamatlépéseket vagy állapotváltozásokat megfelelő elnevezésekkel kell ellátni. Ha minden esetben csak egy bejegyzés (egy sor) van, akkor az adatai nem elég részletesek. Az adatoknak tranzakciós szinten kell lenniük (az egyes esetek előzményeihez hozzá kell férnie), és azokat nem szabad egy esetbe aggregálni. Továbbá legyünk tisztában azzal, hogy a választott tevékenység befolyásolja azt a részletességi szintet, amellyel folyamatunkra tekintünk. Itt is előfordulhat, a tevékenység nevét akár több oszlop kombinációjaként érdemes meghatározni, és több alternatív nézet is lehet arra vonatkozóan, hogy mi minősül tevékenységnek. Végül a folyamatbányászat harmadik fontos előfeltétele, hogy legyen legalább egy **időbélyeg** oszlop, amely jelzi, hogy az egyes tevékenységek mikor történtek. Ez nemcsak a folyamat időzítési viselkedésének elemzéséhez fontos, hanem a tevékenységek sorrendjének megállapításához is az eseménynaplóban. Néha a folyamat minden tevékenységéhez van egy *start* és egy *end* időbélyeg. Ez jó, mert lehetővé teszi egy tevékenység feldolgozási idejének elemzését (az az idő, amelyet valaki aktívan az adott feladat elvégzésére fordított), amelyet végrehajtási időnek vagy tevékenységkezelési időnek is neveznek.



	A	B	C	D	E	F	G
1	CaseID	Timestamp	Medium	Status	Service Line	Urgency	
2	case9700	20.8.09 11:46	Phone	Registered	1st line	0	
3	case9700	20.8.09 11:50	Phone	Completed	1st line	0	
4	case9701	23.9.09 12:23	Phone	Registered	1st line	0	
5	case9701	23.9.09 12:27	Phone	Completed	1st line	0	
6	case9705	20.10.09 14:21	Phone	Registered	Specialist	2	
7	case9705	20.10.09 16:48	Phone	At specialist	Specialist	2	
8	case9705	19.11.09 10:31	Phone	In progress	Specialist	2	
9	case9705	19.11.09 10:32	Phone	Completed	Specialist	2	
10	case3939	15.10.09 11:48	Mail	Registered	Specialist	2	
11	case3939	15.10.09 11:48	Mail	Offered	Specialist	2	
12	case3939	20.10.09 17:18	Mail	In progress	Specialist	2	
13	case3939	20.10.09 17:19	Mail	At specialist	Specialist	2	
14	case3939	21.10.09 14:49	Mail	In progress	Specialist	2	
15	case3939	21.10.09 14:49	Mail	In progress	Specialist	2	
16	case3939	28.10.09 10:17	Mail	In progress	Specialist	2	
17	case3939	28.10.09 10:18	Mail	Completed	Specialist	2	
18	case9704	20.10.09 14:19	Mail	Registered	1st line	0	
19	case9704	20.10.09 14:24	Mail	Completed	1st line	0	
20	case9703	20.10.09 14:40	Phone	Registered	1st line	0	
21	case9703	20.10.09 14:58	Phone	Completed	1st line	0	
22	case9702	24.8.09 12:24	Mail	Registered	2nd line	2	
23	case9702	24.8.09 12:30	Mail	Offered	2nd line	2	
24	case9702	24.8.09 12:31	Mail	Scheduled	2nd line	2	
25	case9702	26.8.09 9:05	Mail	In progress	2nd line	2	
26	case9702	26.8.09 9:19	Mail	Completed	2nd line	2	
27	case9709	20.10.09 14:26	Mail	Registered	Specialist	2	

1. ábra. *Mintanapló fájlra, amely tartalmazza a három kötelező elemet*

2.3.2.1. Az esetazonosítóval kapcsolatban felmerülő problémák:

- Az esetazonosítót több mezőből kell kombinálni: Ez akkor történik, ha a fő esetazonosító önmagában nem azonosítja egyedileg a folyamatpéldányt. Például, ha elemezzük az adóbevallási folyamatot az adóhivatalnál, akkor minden állampolgárt a társadalombiztosítási száma alapján azonosítanak. Az adóbevallásokat azonban minden évben elkészítik. Ha több év adatai vannak, akkor a társadalombiztosítási szám mellett össze kell kapcsolnia azt az évet is, amelyre a nyilatkozatot benyújtották. Az aktivitásnapló készítésekor tehát nem szabad elfeledkezni az egyedi eset azonosításához szükséges összes mezőről.
- Különböző esetazonosítók használata a folyamat különböző részein: Ha a folyamat során több azonosítót használnak, akkor képesnek kell lenni egy eset láncolatának követésére az elejétől a végéig. Például, ha egy vásárlási folyamat során a beérkező kérelmeket PO-számmal azonosítják, és később a fizetési tevékenységek egy számlaszámhoz kapcsolódnak, akkor szükség van arra, hogy a számlaazonosító össze legyen kapcsolva a megrendelés számával (PO), legyen valami referencia, amely segít a folyamatok összerendelésében. Az aktivitás kinyerése során a naplóban egy folyamatos esetazonosítóra van szükség, amely referenciaként szolgál a folyamat minden lépésére.
- Több esetazonosító több-több kapcsolattal rendelkezik: Ha a folyamat során különböző esetazonosítókat használnak, előfordulhat, hogy nem mindig rendelkeznek 1-1 leképezéssel (például két megrendelés kombinálva egy kézbesítésben, vagy egy megrendelés két darabban). Ilyenkor meg kell határozni, hogy melyik szempontból kívánjuk vizsgálni a folyamatot. Előfordulhat, hogy több naplót/adatexportot kell létrehozni, ha több nézőpontot szeretnénk elemezni.

2.3.2.2. Az aktivitással kapcsolatos problémák

A második minimumkövetelmény a naplókkal szemben az említett tevékenység. A tevékenységek a folyamat különböző lépései vagy állapotváltozások. Az informatikai rendszerek nemcsak a számunkra fontos tevékenységeket rögzíthetik, hanem kevésbé érdekes hibakeresési információkat is. Ezért fontos arról meggyőződni, hogy a naplózás a releváns aktivitásokat is rögzíti. A kevésbé releváns tevékenységek naplóban való megléte önmagában nem probléma, később ezek kiszűrhetők.

Nagyon fontos az aktivitások megfelelő elnevezése már a kezdetektől fogva. Ha több jelölt is van a tevékenység elnevezéséhez, akkor célszerű megtartani

mindet, mert az elemzés során így lehetővé válik különböző szempontok vizsgálata. Például egy banki hiteligénylési folyamat során a rendszer rögzítheti mind a belső állapot (a belső folyamat szakaszának bemutatása), mind a külső állapot változását (megmutatja az ügyfél vagy az értékesítési csatorna állapotát). Mindkét szempont hasznos lehet. Ha mindkét mezőt megtartjuk az adatokban, akkor később teljes rugalmasságot élvezhetünk az elemzésben.

Fontos szempont, hogy a naplóban rögzített aktivitások nevei az ember által jól olvashatók legyenek. A legtöbb folyamat összetett, és az elemzés során nem jutunk túl messzire, ha a folyamattérkép lépései pusztán technikai állapotszámokat vagy műveleti kódokat mutatnak. Az adatokban természetesen rögzíthetünk technikai műveleti kódokat is, amelyek később hasznosak lehetnek az adatokkal kapcsolatos technikai kérdések visszakeresésére. Az ember által olvasható név azonban nem elhanyagolható, mert az adja meg pontosan, hogy mi a státusz jelentése. Ezeket az ember által olvasható állapotneveket külön oszlopként kell felvenni az adatokba.

A tevékenységek több helyen rejtve lehetnek. Az adatbázis-központú rendszerekben gyakran a különféle üzleti adattáblákból nyerhetők ki bizonyos tevékenységek. Ebben az esetben automatikusan nem láthatók a folyamat összes eseményei. Ilyenkor meg kell határozni a legfontosabb mérföldkő-tevékenységeket, majd ezek alapján kigyűjteni az aktivitásokat.

2.3.2.3. Az időbélyeggel kapcsolatos problémák

Minden tevékenységhez legalább egy időbélyegre van szükség, hogy az egyes eseményeket megfelelő sorrendbe lehessen rendezni. Ha pedig elemezni szeretnénk a tevékenység időtartamát, akkor minden tevékenységhez meg kell adni egy kezdési és egy befejezési időbélyeget.

Törekedni kell az időbélyegek lehető legpontosabb rögzítésére. Például, ha rendelkezésre állnak az órák és percek adatok is, akkor nem elégséges csupán a dátumot rögzíteni, hanem az időt is célszerű tárolni. Ha pedig másodperc vagy milliszekundum is rendelkezésre áll, az még sikeresebb, még akkor is, ha úgy tűnik, hogy nincs szükség annyi részletre az elemzéshez. A részletes időbélyegek készítése hasznos lehet a tevékenységek sorrendjének megfelelő meghatározásához.

Tevékenységenként több mint két időbélyeg: bizonyos esetekben nemcsak egy vagy két időbélyeg áll rendelkezésre tevékenységenként, hanem több is. Például lehet egy időbélyeg, amely jelzi, hogy egy feladat készen áll-e a felvételre, egy második, amely azt jelzi, hogy az illető mikor kezdett dolgozni rajta, és egy harmadik, amely a feladat befejezését jelzi. Célszerű ezeket mind rögzíteni. A

későbbiekben hasznos lehet különböző megválaszolandó kérdésekben. Például, hogy mennyi az átlagos időtartam a kész és a befejezés között.

A dátum és az idő külön oszlopokban szerepelnek: Mivel egyszerre több időbélyeget is rögzíthetünk, egyetlen időbélyeg dátumának és időpontjának célszerű egy mezőként szerepelnie. Ez segít a későbbi elemzésben.

Különböző időbélyegminták: A különböző adatforrásokból származó időbélyegek formátuma eltérő lehet. Például egy tevékenység dátum-időbélyeg formátuma lehet a 01SEP2013, míg egy másik tevékenységé a 2013-09-10 07.36.51.711899 formátum. A naplóban egységes időbélyeg-formátum alkalmazás az elvárt. Amennyiben többféle formátum is rendelkezésre áll, úgy vagy egységesre hozzuk őket, vagy külön adatként kezeljük.
Időbélyeg-felülírás: Ismétlődő aktivitások során bizonyos időbélyegek felülíródhatnak. Amikor a tevékenység időbélyegeit adatbázisban tároljuk, gyakran feltételezik, hogy a folyamatot az ideális sorrendben hajtják végre, és nem történik újrafeldolgozás. Természetesen ez a legtöbb helyzetben helytelen: Az átdolgozás történhet például azért, mert ezt a feladatot nem először hajtották végre, és a megfelelő folyamatlépést később meg kell ismételni. Ha a rendszer nem rögzíti a tevékenység ismétlésének összes időbélyegét (gyakran csak a legfrissebb időbélyeget őrzi meg), akkor értékes információk vesznek el.

2.3.3. Naplók ismertebb hibái

Bármelyen az inputadatokban megjelenő hiányosság, az úgynevezett „lyukak" alapjaiban befolyásolják a végeredményt. A leggyakrabban felmerülő problémák a következők:

- Formázási hibák: Az egyik leggyakrabban előforduló hiba, hogy bizonyos esetekben a napló adott sorának formátuma eltér az előírttól. Ez nagymértékben befolyásolhatja az elemzés eredményét, jobb esetben csak feldolgozási hibát eredményez. Speciális karakterek, egy felesleges idézőjel számos fejtörést okozhat.
- Hiányzó események: attól még, hogy az adatok hibátlanul importálódtak, továbbra is problémák lehetnek az adatokkal. Egyik tipikus probléma az adatok, események hiánya. Két típusa a következő:

  Lyukak az idővonalban: néha szokatlan hiányosságok fedezhetők fel a napló bizonyos időkeretein belül bekövetkező események számában.

Váratlan adatmennyiség: szükség van arra, hogy nagyjából ismerjük azt, hogy az egyes folyamatok milyen nagyságú tevékenységszámból áll. A váratlan nagy adatmennyiség miatt annak egy része a feldolgozó bizonyos részein elveszhet bármilyen nemű jelzés nélkül egyszerű tervezési hiba miatt.

- Hiányzó aktivitások: Előfordulhat, hogy a folyamat egyes tevékenységeit nem rögzítik az adatok. Lehetnek olyan manuális tevékenységek (például telefonhívások), amelyeket az emberek az asztalunknál végeznek. Ezek a tevékenységek a folyamat során történnek, de nem láthatók az adatokban.
- Hiányzó időbélyeg: Bizonyos helyzetekben tudomásunk van arról, hogy történt-e tevékenység vagy sem, de nincs időbélyeg. Esetleg az aktivitás egy más rendszerből érkezett. Az adatfeldolgozó modul ilyen esetekben nem tudja meghatározni az esemény helyét az idővonalban.
- Befejezetlen folyamatok: a napló olyan folyamatokat is tartalmaz, amelyek rendelkeznek „start" jelzéssel, de „vég" jelzéssel nem. A probléma oka a következőkben keresendő:
  a. Az adatkinyerési módszer csak bizonyos időintervallumon belüli eseményeket vizsgál. Például egyes folyamatok az előző évben kezdődtek (január előtt) és a következő évben folytatódtak. Ebben a helyzetben ezeknek a folyamatoknak csak azt a részét fogja látni, amelyek abban az évben történtek, amelyet elemez.
  b. Néhány folyamat még nem fejeződött be. Még akkor is, ha minden adatot kinyertünk, előfordulhat, hogy az esetek egy része még nem fejeződött be. Még mindig „valahol a közepén vannak".
  c. Egyes folyamatok soha nem fejeződnek be. Nem minden folyamat fog rendelkezni „vég" jelzéssel. Bizonyos esetekben, akár technikai hiba miatt a folyamat megszakad. Például egy webshopban elvégzett fizetés után az ügyfél nem irányítódik vissza a bolt felületére. Ezek az esetek nem érnek véget a várható végpontok egyikén sem, soha nem fejeződnek be.

2.4. Problémamegoldás neurális hálózattal

A mesterséges neurális hálózatok napjaink egyik leginkább kutatott területe. Segítségével olyan problémák válnak megoldhatóvá, amelyek korábban nem, vagy csak részlegesen. Természetesen a hardverek fejlődése ehhez nagy támpontot nyújtott. Ma már számos modellt tart nyilván az irodalom, a leghatékonyabb megközelítések gyakran az úgynevezett mély tanulás (deep learning) területhez kapcsolódnak. A mesterséges neurális hálózatok tanítása minden esetben egy felügyelt regressziós problémára kerül visszavezetésre, de kivitelezhető osztályozás

és nem felügyelt tanítás is. A hálózatok működésében két fázist különíthetünk el: tanítási fázisban a ismert bemeneti paraméterek és várt kimenetek ismeretében a súlyokat változtatjuk úgy, hogy egy veszteségfüggvény értékét (például az átlagos négyzetes hibát) minimalizáljuk ezzel. A feltanított neurális hálózat a predikciós fázisban ezután ismeretlen bemenet átadásakor kimenetet képez, mely lehet például egy kategóriába való tartozás valószínűsége.

Neurális hálózatokat a gyakorlatban előszeretettel alkalmazzák a különböző aktivitások előrejelzésére. A tevékenységnapló adatai alapján a műveletsori szekvenciák azonosíthatók, valamint az aktuális tevékenységsor pedig illeszthető a meglévő mintákra. A folyamatfeltárás neurális hálózat alapú megközelítése azonban számos problémát vet fel. Semmiféleképpen sem nevezhető tipikus ANN-feladatnak, mert a probléma megoldása egy nem felügyelt tanulást kíván meg, a neurális hálózatok pedig főként felügyelt tanulásra alkalmasak. Ezen a területen számos kutatási irány/probléma fog előtérbe kerülni, amelyek arra a kérdésre fogják keresni a választ, hogy miként lehet megalkotni egy olyan hálózatot, amely képes az említett problémát megoldani. A jövőben új, dinamikus struktúrájú neurálisháló-modell kidolgozása fogják a kutatások alapját képezni.

2.4.1. Tanítóminta

A neurális hálózatok tanulási folyamatához tanítómintára van szükség. A korábban említésre került inputadatok hiánya/nem megfelelő minősége alapjaiban befolyásolhatja a tanulás folyamatát. Az inputadathalmaznak kellően komplexnek kell lennie – tartalmazva az összes előforduló folyamatot, azok aktivitásait – ahhoz, hogy a különböző megközelítések, hipotézisek és modellek maradéktalanul kipróbálhatók legyenek a modellalkotás során. Tovább kimondható, hogy egyetlen mintaadathalmaz önmagában szintén nem elegendő, különben a kidolgozott modell nem tesztelhető hatékonyan a valóságban előforduló sokszínűség kezelésében.

2.4.2. Erőforrás-szükséglet

Az utóbbi években egyre hangsúlyosabb szerepet kapnak az úgynevezett LSTM típusú neurális hálózatok, melyek számtalan helyen (Siri, Cortana, Google Voice Assistant, Alexa) bizonyították hatékonyságukat. Bár hatékonyak, célszerű odafigyelni a hátrányukra is:

Az LSTM-alapú hálózatok tanítási folyamata nehezebb, mert memória-sávszélességhez megfelelően igazított számítást igényelnek, ami a hardvertervezők szá-

mára egy külön fejtörés, valamint korlátozza a hálózat gyakorlati alkalmazhatóságát. Az LSTM-nek cellánként 4 lineáris rétegre (MLP-réteg) van szüksége ahhoz, hogy az egyes szekvencia-idő-lépésekben és az egyes szakaszokban fusson. A lineáris rétegek nagy mennyiségű memória sávszélességet igényelnek, ebből kifolyólag gyakran nem használhatnak sok számítási egységet, mert a rendszer nem rendelkezik elegendő memória-sávszélességgel a számítási egységek "táplálásához". További számítási egységeket hozzáadni könnyű, azonban memória-sávszélességét növelni sokkal nehezebb. Ennek eredményeként az RNN / LSTM és a variánsok nem gyorsíthatók hardveresen megfelelően.

Tehát LSTM alapú hálózatokkal való dolgozás mindenféleképpen nagyobb erőforrás-szükségletet eredményez, amely főként a tanulási fázist érinti. Ráadásul a projekt kapcsán kezelt, illetve várt inputként szolgáló adathalmaz is nagy. Ki kell dolgozni tehát egy olyan környezetet az kutatók számára, ahol osztott módon képesek kihasználni a rendelkezésre álló szerverkapacitásokat. A mintarendszer rendelkezzen elegendő memória-, számítási és tárkapacitással. Így az oktatók nem a saját (általában alacsony teljesítményű) gépüket használják a komplexebb tanítási folyamatok tesztelésére, hanem egy sokkal gyorsabban eredményt szolgáltató környezetet.

3. Összegzés

A modern vállalatok ügyviteli folyamataik egyre komplexebbek, melyeket egyre összetettebb rendszerekkel lehet támogatni. Természetesen, mint minden más területhez hasonlóan, itt is felmerül az automatizálhatóság kérdése, amely segítségével a bonyolult folyamatok részben egyszerűsíthetők. Az RPA éppen egy ilyen kezdeményezés, amely során a felhasználói aktivitás naplózással és figyelésével felépíthető egy olyan automatizált döntéstámogató rendszer, amely képes a komplex folyamat bizonyos lépéseinek megoldására, vagy akár előrejelzésére. A terület meglehetősen modern volta miatt még nem kiforrott, nem állnak rendelkezésre dobozos termékek a megvalósításra. Jelen publikációban összefoglaltuk a terület legfontosabb követelményeit és jellegzetes problémáit, melyek jó alapot nyújtanak egy ilyen jellegű rendszer megvalósításához.

ESEMÉNYNAPLÓ-LEÍRÓ SZABVÁNYOK ELEMZÉSE

BAKSÁNÉ VARGA ERIKA

A projekt fő célja, hogy neurális háló alapú megoldást találjunk a gyakori folyamatszekvenciák és folyamatgráfok meghatározására az eseménynapló állományok adatai alapján. A folyamatbányász algoritmusok alkalmazása előtt azonban biztosítanunk kell az eseménynaplók megfelelő minőségét és egységes formátumát. Jelen kutatás célja megtalálni azt a formátumot, amelyre a meglévő eseménynaplók egyszerűen konvertálhatók, illeszkedik a neurális hálók inputformátumára, és amely biztosítja a szekvenciafeldolgozó algoritmusok hatékony működését.

1. A kutatás célja és lépései

A számítógépen végrehajtott ügyviteli folyamatokat eseménynaplókban tárolják. Ahhoz, hogy az eseménynaplókból ki tudjuk nyerni a leggyakoribb folyamatokat, vagy előre tudjuk jelezni a soron következő folyamatelemet, folyamatbányász eljárásokat kell implementálni. Mivel ezeknek az algoritmusoknak a bemenete eseménynapló, lényeges hogy specifikáljuk ezek formátumát.

Az eseménynapló formátuma az eseményeket rögzítő szoftvertől függ. A Unix-alapú operációs rendszerek például .log kiterjesztésű olvasható állományokban naplózzák az eseményeket, ahol minden sor egy adott esemény bekövetkezésének időpontját és leírását tartalmazza. A Windows operációs rendszernek a 7-es verziótól kezdve saját XML-alapú eseménynapló-leíró formátuma van (EVTX) (korábbi verziókban EVT). Az .evtx kiterjesztésű bináris állományok csak az Event Viewer programmal tekinthetők meg. Mi pedig a kutatás kísérleti fázisában a Task Force on Process Mining oldalról (www.tf-pm.org) letölthető, folyamatbányászathoz előkészített CSV- és XES-állományokkal dolgoztunk.

Érezhető tehát a formátum egységesítésének igénye. Ebben a kutatási jelentésben bemutatom a rendelkezésre álló eseménynapló-leíró szabványokat és megvizsgálom a különböző formátumok közötti konverzió lehetőségét és lépéseit.

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

A vizsgált eseménynapló-formátumok: txt, csv, json, evtx, xml, xes, ocel és spmf. Megvizsgáltam ezek között a konverzió lehetőségét azzal a céllal, hogy kijelöljem azt a formátumot, amelyre minden állományt konvertálni fogunk a folyamatbányászat megkezdése előtt.

2.1.1. Eseménynaplók konvertálása XES szabványos leírásra

Az XES (eXtensible Event Stream) szabvány egy egységes XML-alapú nyelvet definiál az esemény adatok tárolásához, továbbításához és feldolgozásához. A szabvány két XML Schema leírást tartalmaz: az egyik az XES-eseménynaplók szerkezetét definiálja, a másik egy ilyen naplóhoz tartozó kiterjesztés definícióját határozza meg (https://xes-standard.org/).

Az XES-dokumentum szerkezete:

```
<log  xes.version="1.0"  xes.features="..." xmlns="http://www.xes-standard.org/">
      <trace>
            <event>
                  …
            </event>
      </trace>
</log>
```

Az eseménynapló (log) a végrehajtott folyamatok (trace) tárolására szolgál. Az egyes folyamatokon belül tetszőleges számú esemény (event) fordulhat elő. A log, trace és event jelölők önmagukban nem hordoznak információt, csak az eseménynapló struktúráját definiálják. Az információkat ezek attribútumai tárolják kulcs-érték (key-value) párokban. Hat elemi attribútum létezik, amelyeket az általuk reprezentált adatérték típusa határoz meg (String, Date, Int, Float, Boolean és ID); és két összetett attribútum: a rendezett lista (list) és a halmaz (container). Az összetett attribútumok azonban csak akkor használhatók, ha az „attribútumok egymásba ágyazása" tulajdonság használatát az eseménynapló elején jelezzük a <log> jelölő tagban (*<log xes.version="1.0" xes.features="nested-attributes">*) és olyan XES-feldolgozót alkalmazunk, amely támogatja ezt a tulajdonságot.

Az XES-szabványban egyik struktúrajelölő elemnek (log, trace, event) sincs előre meghatározott attribútumlistája. Ennek következtében az egyes attribútumok jelentése sincs előre rögzítve, nem egyértelmű. Ezt a hiányosságot küszöbölhetjük ki a kiterjesztések használatával. Az első XES szabvány hét kiterjesztést definiál, amelyek XESEXT XML formátumban vannak megadva: concept, cost, id, lifecycle, organizational, semantic, time. Az újabb, IEEE 1849-2016 szabvány további öt kiterjesztést definiál: artifactlifecycle, micro, swcomm, swevent, swtelemetry.

2.1.1.1. TXT- és CSV-napló átalakítása XES-formátumra

A szöveges naplóállomány tartalma az XES log jelölőelemek közé kerül. Az állomány minden egyes sora külön eseményt (event) ír le, a bekövetkezésük sorrendjében.

Ezeknek az eseményeknek egy adott folyamathoz tartozását vagy nem tárolják, vagy az eseményleírók között szerepel egy folyamatazonosító. Amikor XES-formátumra konvertáljuk a naplót, azokat az eseményeket csoportosítjuk egy tracebe, amelyeknél a folyamatazonosító megegyezik; és ez az adat a trace attribútuma lesz. A folyamatazonosítón kívül is lehetnek olyan eseményleírók, amelyek azonos értékkel rendelkeznek a folyamathoz tartozó összes eseményben. Ezek is a trace attribútumai lesznek az XES-naplóban. Ezeken felül a további eseményleírók az esemény attribútumai.

Amennyiben az események folyamathoz tartozásáról nincs információnk, kétféle feltételezéssel élhetünk. A konkrét naplózási stratégia ismeretében mondhatjuk azt, hogy minden esemény önálló folyamat (ekkor az XES-naplóban minden traceben egyetlen event tag lesz); vagy tekinthetünk úgy egy naplóra, hogy az egy adott folyamathoz tartozó eseményeket tárolja (ekkor az XES lognak csak egyetlen trace tagja van és minden event ezen belül kap helyet).

Az XES-naplóban az attribútumokat kulcs-érték párokkal adjuk meg. A kulcsokat a TXT- / CSV-állomány fejlécének (első sorának) kellene tartalmaznia. Amennyiben ez hiányzik, úgy generált kulcsokat alkalmazunk. Az XES-napló az attribútumok típusát is tartalmazza. A legtöbb esetben ez az információ nem nyerhető ki a TXT- / CSV-állományból. Ekkor a legáltalánosabb "szöveg" típust használjuk.

2.1.1.2. JSON-napló átalakítása XES-formátumra

A JSON-(JavaScript Object Notation, JavaScript objektumjelölés) programok közötti strukturált adatcserét lehetővé tevő, szabványos és viszonylag egyszerű szerkezetű formátum. A JSON-fájl szövegfile, tartalma kapcsos zárójel párba zárt vesszővel elválasztott kulcs-érték párosok, amelyek között : van. A kulcs string, és egyedinek kell lennie. Az érték lehet konstans; kapcsos zárójelek közé zárt kulcs-érték párok (asszociatív tömb, amit szokás „objektumnak" is nevezni), vagy szögletes zárójelek közé zárt kulcs-érték párok (rendezett tömb). Ezek az elemek egymásba ágyazhatók: egy objektum elemként tartalmazhat egy másik objektumot vagy tömböt, a tömb elemei is lehetnek objektumok vagy tömbök. Így egészen bonyolult adatszerkezetek is leírhatók.

Egy JSON-naplófájlban egy sorban egy esemény leírása található a megadott formában (JSONLines, vagy JSONL formátum) (https://jsonlines.org/). Amikor XES-formátumra alakítjuk, a JSON kulcs-érték párokat leképezzük XES-esemény attribútum-érték párokra. Annak a megállapítása viszont, hogy egy adott esemény melyik folyamathoz tartozik (trace), ugyanúgy történik mint a szöveges naplóállományok esetén. Vagy az eltárolt adatok, vagy a naplózási stratégia alapján döntünk.

2.1.1.3. EVTX és XML napló átalakítása XES formátumra

Az EVTX- (https://www.sciencedirect.com/science/article/pii/S1742287607000424) napló két lépcsőben alakítható át XES-formátumra. Első lépésben az EvtxParser alkalmazásával XML-formátumra alakítjuk. Az XML-formátumú napló szerkezetét az alábbi ábra mutatja:

```
<Events>
  <Event>
    <System>...</System>
    <EventData>...</EventData>
  </Event>
  <Event>
    ...
  </Event>
</Events>
```

Minden esemény leírása a System elemmel kezdődik. Olyan alapvető adatokat tárol el itt automatikusan a rendszer, mint például az esemény bekövetkezésének helye (a számítógép IP-címe), az esemény időbélyege, vagy az esemény azonosítószáma. Emellett az alábbi elemek közül szerepel még egy az esemény leírásánál: BinaryEventData, DebugData, EventData, ProcessingErrorData, RenderingInfo, vagy UserData. Leggyakrabban az EventData elemmel találkozhatunk, ami az eseményt kiváltó alkalmazás által átadott paramétereket tartalmazza.

Látható, hogy ez a formátum is az események szekvenciáját tartalmazza, ugyanúgy mint a szöveges leíró állományok. Az események XES-folyamatokhoz (trace) rendelését az eltárolt adatok, vagy a naplózási stratégia alapján határozhatjuk meg. Viszont az XML event tag egy az egyben megfeleltethető az XES event tagnak.

2.1.2. XES-formátum átalakítása OCEL szabványos leírásra

Látható, hogy az XES folyamatcentrikus leírásmód, azaz központi eleme a trace, és az eseménynaplóban rögzített minden tevékenységet egy folyamathoz kötünk. Az információs rendszerekben azonban, a párhuzamosan zajló folyamatok során egy adott esemény rendszerint több, különböző folyamatban érintett objektumra vonatkozik. Azaz az XES-szabvány szerinti folyamatcentrikus modellezéskor az adott eseményt egyidejűleg csak egy objektumtípus szemszögéből vizsgáljuk. Ezzel viszont információt vesztünk. Ezért dolgozták ki az OCEL (Object-Centric Event Logs) objektumcentrikus eseménynapló-formátumot (http://www.ocel-standard.org/).

Az OCEL az objektumcentrikus eseménynaplók szabványos leíró nyelve. Szintaktikáját tekintve nagyon hasonlít az XES-szabványhoz. A legfontosabb különbségek a két nyelv között:

- Az OCEL nyelv nem használja a „trace" (folyamatpéldány) jelölőelemet.
- Az OCEL bevezeti az objektum („object") jelölőelemet.

Az OCEL-naplóállomány megadható XML- és JSON-formátumban is. Az elején globális listában definiáljuk a naplóban szereplő objektumtípusokat és az eseményeket leíró attribútumokat. Ezután következik a bekövetkezett események leírása, ami az eseményattribútumok értékét és az érintett objektumok referenciáját tartalmazza.

Az XES-formátum OCEL-leírásra konvertálásakor a trace jellemzői az események attribútumai között fognak szerepelni. Az XES-eseményattribútumokból osztályokat képzünk. Ezekből a típusokból lesznek az objektumok, amelyeket külön definiálunk és az adott esemény csak a referenciájukat tárolja. Azok az XES-eseményattribútumok, amelyek közvetlenül az eseményhez kapcsolódnak (nem pedig az esemény által érintett valamely objektumhoz), továbbra is eseményattribútumok lesznek.

2.1.3. Szöveges napló átalakítása SPMF-formátumra

Az SPMF (Sequence Pattern Mining Format) (http://www.philippe-fournier-viger.com/spmf/) egy Java könyvtár, ami elemhalmazok és szekvenciák bányászatát támogató algoritmusok implementációját foglalja magában. Az input adatállományokhoz saját formátumot (és erre a formátumra konvertáló eljárásokat) biztosít. Az SPMF formátumban minden sor egy esemény leírása számsorozattal. A számok az eseményhez tartozó attribútum-érték párok sorszámai. Az attribútum-érték párok metaadatok, azaz az adatbányász algoritmusok nem használják fel ezeket, csak a kapott eredmények értelmezhetősége miatt tartoznak hozzá a formátumleíráshoz.

A szöveges naplóállomány átalakítása során egy SPMF-szekvencia-adatbázist hozunk létre. Ez tartalmazza az attribútum-érték párok definícióját és az eseményeket leíró szekvenciákat. Az attribútum-érték párok definíciós sora @ITEM = sorszám = érték alakú, ahol a sorszám 1-től kezdődően folytonosan növekszik és minden egyedi értékre különböző. Egy esemény adatainak definícióját követően felírjuk az eseményt, mint az egymás után következő értékek szekvenciáját, ahol az egyes értékek között –1 az elválasztó karakter, és a szekvencia végét –2 zárja. Ennek a formátumnak az előnye az információtárolás minimalizálása és ezáltal az adatbányász algoritmusok hatékonyságának a biztosítása.

2.2. Kiértékeléshez használt referenciafolyamatok és értékek bemutatása

Az ügyviteli folyamatokkal kapcsolatban eltárolt adatok:

case id	customer request			
id	id	description	priority	source

activity						agent			
id	time	state	type	cate-gory	input	id	group	role	host

Adatelem	**Leírás**
Case id	az ügyviteli folyamat egyedi azonosítója (egy folyamat több tevékenységet foglalhat magában)
Request id	a beérkezett kérés azonosítója (előfordulhat, hogy egy kérés kiszolgálása több ügyviteli folyamatot is elindít)
Request description	a kérés szöveges leírása
Request priority	a kérés prioritása, sürgőssége (előre definiált prioritási szintek alapján)
Request source	a kérés forrása (az ügyfél azonosítását szolgáló adatok)
Activity id	az ügyviteli folyamat során elvégzett tevékenység azonosítója (előre definiált tevékenységlista alapján)
Activity time	a tevékenység végrehajtási ideje (rendszer időbélyeg)
Activity state	a tevékenység állapota (pl.: befejezett, felfüggesztett, ütemezett stb.) (előre definiált állapotlista alapján)
Activity type / category	előre definiált tevékenységhierarchia alapján a tevékenység besorolása (típusa, kategóriája)
Activity input	a tevékenység során rögzített adatérték (ha volt)
Agent id	a tevékenységet végrehajtó ügyintéző azonosítója
Agent group	a tevékenységet végrehajtó ügyintéző melyik szervezeti egységhez tartozik
Agent role	a tevékenységet végrehajtó ügyintéző munkaköre
Agent host	a tevékenység végrehajtásának helye (a számítógép azonosítója)

A különböző formátumok közötti konverziós eljárások kidolgozása során felhasznált naplóállományok, amelyek egy példa ügyviteli folyamat 4 eseményének adatait tartalmazzák:

TXT, CSV:

```
Case-id;Request-id;Request-desc;Request-priority;Request-source;Activity-id;Activity-time;Activity-state;Activity-type;Activity-category;Activity-input;Agent-id;Agent-group;Agent-role;Agent-host
111;222;invoice complaint;urgent;333;441;2021-11-07T02:26:35.000+01:00;completed;register request;invoicing;;555;customer service;clerk;193.4.5.6
111;222;invoice complaint;urgent;333;442;2021-11-08T02:26:35.000+01:00;completed;examine request;invoicing;invoice data;666;accounting;clerk;194.6.5.8
111;222;invoice complaint;urgent;333;443;2021-11-10T02:26:35.000+01:00;completed;decide;invoicing;decision;777;accounting;administrator;195.4.5.6
111;222;invoice complaint;urgent;333;444;2021-11-11T02:26:35.000+01:00;completed;reject request;invoicing;;555;customer service;clerk;193.4.5.6
```

JSONL:

{"Activity-category":"invoicing","Activity-id":441,"Activity-input":"","Activity-state":"completed","Activity-time":"2021-11-07T02:26:35.000+01:00","Activity-type":"register request","Agent-group":"customer service","Agent-host":"193.4.5.6","Agent-id":555,"Agent-role":"clerk","Case-id":111,"Request-desc":"invoice complaint","Request-id":222,"Request-priority":"urgent","Request-source":333}
{"Activity-category":"invoicing","Activity-id":442,"Activity-input":"invoice data","Activity-state":"completed","Activity-time":"2021-11-08T02:26:35.000+01:00","Activity-type":"examine request","Agent-group":"accounting","Agent-host":"194.6.5.8","Agent-id":666,"Agent-role":"clerk","Case-id":111,"Request-desc":"invoice complaint","Request-id":222,"Request-priority":"urgent","Request-source":333}
{"Activity-category":"invoicing","Activity-id":443,"Activity-input":"decision","Activity-state":"completed","Activity-time":"2021-11-10T02:26:35.000+01:00","Activity-type":"decide","Agent-group":"accounting","Agent-host":"195.4.5.6","Agent-id":777,"Agent-role":"administrator","Case-id":111,"Request-desc":"invoice complaint","Request-id":222,"Request-priority":"urgent","Request-source":333}
{"Activity-category":"invoicing","Activity-id":444,"Activity-input":"","Activity-state":"completed","Activity-time":"2021-11-11T02:26:35.000+01:00","Activity-type":"reject request","Agent-group":"customer service","Agent-host":"193.4.5.6","Agent-id":555,"Agent-role":"clerk","Case-id":111,"Request-desc":"invoice complaint","Request-id":222,"Request-priority":"urgent","Request-source":333}

XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <row>
    <Case-id>111</Case-id>
    <Request-id>222</Request-id>
    <Request-desc>invoice complaint</Request-desc>
    <Request-priority>urgent</Request-priority>
    <Request-source>333</Request-source>
    <Activity-id>441</Activity-id>
    <Activity-time>2021-11-07T02:26:35.000+01:00</Activity-time>
    <Activity-state>completed</Activity-state>
```

```
  <Activity-type>register request</Activity-type>
  <Activity-category>invoicing</Activity-category>
  <Activity-input></Activity-input>
  <Agent-id>555</Agent-id>
  <Agent-group>customer service</Agent-group>
  <Agent-role>clerk</Agent-role>
  <Agent-host>193.4.5.6</Agent-host>
</row>
<row>
  <Case-id>111</Case-id>
  <Request-id>222</Request-id>
  <Request-desc>invoice complaint</Request-desc>
  <Request-priority>urgent</Request-priority>
  <Request-source>333</Request-source>
  <Activity-id>442</Activity-id>
  <Activity-time>2021-11-08T02:26:35.000+01:00</Activity-time>
  <Activity-state>completed</Activity-state>
  <Activity-type>examine request</Activity-type>
  <Activity-category>invoicing</Activity-category>
  <Activity-input>invoice data</Activity-input>
  <Agent-id>666</Agent-id>
  <Agent-group>accounting</Agent-group>
  <Agent-role>clerk</Agent-role>
  <Agent-host>194.6.5.8</Agent-host>
</row>
<row>
  <Case-id>111</Case-id>
  <Request-id>222</Request-id>
  <Request-desc>invoice complaint</Request-desc>
  <Request-priority>urgent</Request-priority>
  <Request-source>333</Request-source>
  <Activity-id>443</Activity-id>
  <Activity-time>2021-11-10T02:26:35.000+01:00</Activity-time>
  <Activity-state>completed</Activity-state>
  <Activity-type>decide</Activity-type>
  <Activity-category>invoicing</Activity-category>
  <Activity-input>decision</Activity-input>
  <Agent-id>777</Agent-id>
  <Agent-group>accounting</Agent-group>
  <Agent-role>administrator</Agent-role>
  <Agent-host>195.4.5.6</Agent-host>
</row>
<row>
```

```
    <Case-id>111</Case-id>
    <Request-id>222</Request-id>
    <Request-desc>invoice complaint</Request-desc>
    <Request-priority>urgent</Request-priority>
    <Request-source>333</Request-source>
    <Activity-id>444</Activity-id>
    <Activity-time>2021-11-11T02:26:35.000+01:00</Activity-time>
    <Activity-state>completed</Activity-state>
    <Activity-type>reject request</Activity-type>
    <Activity-category>invoicing</Activity-category>
    <Activity-input></Activity-input>
    <Agent-id>555</Agent-id>
    <Agent-group>customer service</Agent-group>
    <Agent-role>clerk</Agent-role>
    <Agent-host>193.4.5.6</Agent-host>
  </row>
</root>
```

2.3. Kiértékelések eredményeinek bemutatása

A következő táblázatok a formátumátalakítási lépéseket foglalják össze.

TXT, CSV, JSONL, XML → XES

A vastagon szedett XES-attribútumkulcsok szabványos kiterjesztések, amelyek alkalmazása segíti az egységes szemantikai értelmezést. Az azonos színnel megjelölt attribútumokból pedig összetett attribútumok alkothatók, ami támogatja az objektumközpontú ábrázolást.

Forrás	XES bennfoglaló tag	XES-attribútum
Case id	<trace>	<string key="**concept:name**" value="111"/>
Request id	<trace>	<string key="request-id" value="222">
Request description	<trace>	<string key="request-desc" value="invoice complaint">
Request priority	<trace>	<string key="request-priority" value="urgent">
Request source	<trace>	<string key="request-source" value="333">
Activity id	<event>	<string key="**concept:name**" value="441"/>
Activity time	<event>	<date key="**time:timestamp**" value="2009-11-25T12:45:32.345+02:00" />
Activity state	<event>	<string key="**lifecycle:transition**" value="completed"/>

Forrás	XES bennfoglaló tag	XES-attribútum
Activity type	<event>	<string key="activity-type" value="register request" />
Activity category	<event>	<string key="activity-category" value="invoicing" />
Activity input	<event>	<string key="activity-input" value="" />

Forrás	XES bennfoglaló tag	XES-attribútum
Agent id	<event>	<string key="**org:resource**" value="555" />
Agent group	<event>	<string key="**org:group**" value="customer service" />
Agent role	<event>	<string key="**org:role**" value="clerk" />
Agent host	<event>	<string key="**agent-host**" value="193.4.5.6" />

XES → OCEL

OCEL-ben egy esemény leírása az eseményattribútumok értékét és az érintett objektumok referenciáját tartalmazza. Az ugyanahhoz a folyamathoz tartozó eseményeket az <events> ... </events> jelölő elemek között adjuk meg. Az XES-folyamatazonosítót bele kell kódolni az események azonosítójába.

Forrás	OCEL-esemény	OCEL-leírás
Case id + Activity id	<string key="name" value="id"/>	<string key="id" value="111-441"/>
Event time	<string key="name" value="time"/>	<date key="timestamp" value="2009-11-25T12:45:32.345+02:00" />
Event state	<string key="name" value="state"/>	<string key="state" value="completed"/>

Forrás	OCEL-objektum	OCEL-leírás
Request id	<string key="type" value="request"/>	<string key="id" value="222"/>
Request description	<string key="type" value="request"/>	<string key="type" value="invoice complaint"/>
Request priority	<string key="type" value="request"/>	<string key="priority" value="urgent" />
Request source	<string key="type" value="request"/>	<string key="source" value="333" />
Activity id	<string key="type" value="activity"/>	<string key="id" value="222"/>

Forrás	OCEL-objektum	OCEL-leírás
Activity type	<string key="type" value="activity"/>	<string key="type" value="register request" />
Activity category	<string key="type" value="activity"/>	<string key="category" value="invoicing" />
Activity time	<string key="type" value="activity"/>	<date key="timestamp" value="2009-11-25T12:45:32.345+02:00" />
Activity state	<string key="type" value="activity"/>	<string key="state" value="completed"/>
Activity input	<string key="type" value="activity"/>	<string key="input" value=""/>
Agent id	<string key="type" value="agent"/>	<string key="id" value="555"/>
Agent role	<string key="type" value="agent"/>	<string key="type" value="clerk" />
Agent group	<string key="type" value="agent"/>	<string key="group" value="customer service" />
Agent host	<string key="type" value="agent"/>	<string key="host" value="193.4.5.6"/>

TXT, CSV → SPMF

Az SPMF-szekvencia-adatbázisban metaadat-definíciók és szekvenciák követik egymást. Az üresen hagyott mezők figyelembevételéhez, illetve azért, hogy minden szekvencia azonos hosszúságú legyen, bevezettem a helykitöltő „null” értéket.

@ITEM=1=111
@ITEM=2=222
@ITEM=3=invoice complaint
@ITEM=4=urgent
@ITEM=5=333
@ITEM=6=441
@ITEM=7=2021-11-07T02:26:35.000+01:00
@ITEM=8=completed
@ITEM=9=register request
@ITEM=10=invoicing
@ITEM=11=null
@ITEM=12=555
@ITEM=13=customer service
@ITEM=14=clerk
@ITEM=15=193.4.5.6
1 -1 2 -1 3 -1 4 -1 5 -1 6 -1 7 -1 8 -1 9 -1 10 -1 11 -1 12 -1 13 -1 14 -1 15 -1 -2

2.4. Eredményeket szemléltető képernyőképek

A minta szöveges naplóállomány első sorának XES-szabvány szerinti leírása:

```
<?xml version="1.0" encoding="UTF-8" ?>
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7">
	<extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
	<extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.xesext"/>
	<extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
	<extension name="Organizational" prefix="org" uri="http://code.fluxicon.com/xes/org.xesext" />
	<string key="concept:name" value="XES Event Log"/>
	<trace>
		<string key="concept:name" value="111"/>
		<string key="request" value="222">
			<string key="description" value="invoice complaint" />
			<string key="priority" value="urgent" />
			<string key="source" value="333" />
		</string>
		<event>
			<string key="concept:name" value="441"/>
			<date key="time:timestamp" value="2009-11-25T12:45:32.345+02:00" />
			<string key="lifecycle:transition" value="completed"/>
			<string key="type" value="register request" />
			<string key="category" value="invoicing" />
			<string key="input" value="" />
			<string key="agent" value="555">
				<string key="org:group" value="customer service" />
				<string key="org:role" value="clerk" />
				<string key="host" value="193.4.5.6" />
			</string>
		</event>
	</trace>
</log>
```

A minta szöveges naplóállomány első sorának OCEL-szabvány szerinti leírása:

```
<?xml version='1.0' encoding='UTF-8'?>
<log>
 <global scope="event">
  <string key="id" value="__INVALID__"/>
  <string key="timestamp" value="__INVALID__">
  <string key="state" value="__INVALID__"/>
  <string key="omap" value="__INVALID__"/>
 </global>
 <global scope="object">
  <string key="id" value="__INVALID__"/>
  <string key="type" value="__INVALID__"/>
 </global>
 <global scope="log">
  <string key="version" value="0.1" />
  <string key="ordering" value="timestamp" />
  <list key="attribute-names">
   <string key="name" value="request-priority"/>
   <string key="name" value="request-source"/>
   <string key="name" value="activity-category"/>
```

```
    <string key="name" value="activity-time"/>
    <string key="name" value="activity-state"/>
    <string key="name" value="activity-input"/>
    <string key="name" value="agent-group"/>
    <string key="name" value="agent-host"/>
   </list>
   <list key="object-types">
    <string key="type" value="request"/>
    <string key="type" value="activity"/>
    <string key="type" value="agent"/>
   </list>
  </global>
  <events>
   <event>
    <string key="id" value="111-441"/>
    <date key="timestamp" value="2020-07-09T08:20:01.527+01:00"/>
    <string key="state" value="completed"/>
    <list key="omap">
     <string key="object-id" value="222"/>
     <string key="object-id" value="441"/>
     <string key="object-id" value="555"/>
    </list>
   </event>
   ...
  </events>
 <objects>
   <object>
    <string key="id" value="222"/>
    <string key="type" value="invoice complaint"/>
    <list key="ovmap">
        <string key="request-source" value="333" />
         <string key="request-priority" value="urgent" />
    </list>
   </object>
   <object>
    <string key="id" value="441"/>
    <string key="type" value="register request"/>
    <list key="ovmap">
       <string key="activity-category" value="invoicing" />
       <date key="activity-time" value="2020-07-09T08:20:01.527+01:00"/>
       <string key="activity-state" value="completed"/>
      <string key="activity-input" value=""/>
    </list>
   </object>
   <object>
    <string key="id" value="555"/>
    <string key="type" value="clerk"/>
    <list key="ovmap">
       <string key="agent-group" value="customer service" />
       <string key="agent-host" value="193.4.5.6" />
    </list>
   </object>
  </objects>
</log>
```

A teljes minta szöveges naplóállomány SPMF-formátum szerinti szekvencia leírása:

1 -1 2 -1 3 -1 4 -1 5 -1 6 -1 7 -1 8 -1 9 -1 10 -1 11 -1 12 -1 13 -1 14 -1 15 -1 -2
1 -1 2 -1 3 -1 4 -1 5 -1 16 -1 17 -1 8 -1 18 -1 10 -1 19 -1 20 -1 21 -1 14 -1 22 -1 -2
1 -1 2 -1 3 -1 4 -1 5 -1 23 -1 24 -1 8 -1 25 -1 10 -1 26 -1 27 -1 21 -1 28 -1 29 -1 -2
1 -1 2 -1 3 -1 4 -1 5 -1 30 -1 31 -1 8 -1 32 -1 10 -1 11 -1 12 -1 13 -1 14 -1 15 -1 -2

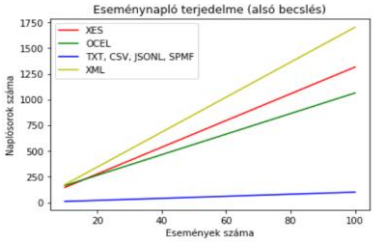
2.5. Eredményeket szemléltető diagramok

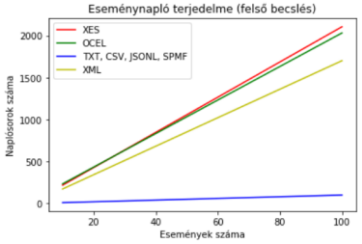
Az XES-, az OCEL- és az SPMF-formátumok szerint leírt eseménynaplókat a napló terjedelme (L) alapján hasonlítottam össze. A napló terjedelme az állomány hossza (a kódsorok száma), amelyet a tárolt információelemek számának függvényeként határoztam meg. Ez a mérőszám azért lényeges, mert befolyásolja a naplófeldolgozó eljárások futási sebességét.

L – terjedelem
FIX – a dokumentum információelemektől független sorainak száma
T – folyamatok száma (trace)
TA – folyamat attribútumainak száma
E – események száma
EA – események attribútumainak száma
O – objektumok száma Opreambulum hossza PRE

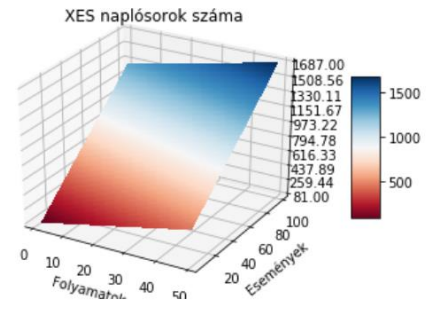
Dokumentum	**Terjedelem**	**ERPA alsó becslés**	**ERPA felső becslés**
XES	FIX + (T*TA) + (E*EA)	T = 1 8 + 8 + E*13	T = E 8 + E*8 + E*13
OCEL	FIX + EA + (E*10) + (O*EA)	O = 3 34 + E*10 + 30	O = E 34 + E*10*2
TXT, CSV, JSONL, SPMF	E	E	E
XML	FIX + E * (TA + EA + 2)	3 + E*17	3 + E*17

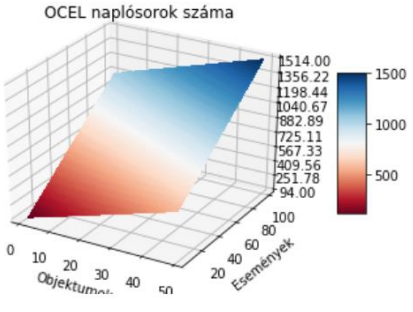
A terjedelem növekedésének szemléltetése csak az események számának függvényében:





A terjedelem növekedése mindkét változó figyelembevétele mellett:





Az ábrából látható, hogy az objektumok számának növekedése nagyobb mértékben hozzájárul a terjedelem növekedéséhez, mint a folyamatok számának növekedése. Az események leírása viszont hosszabb a folyamatcentrikus megközelítésben, mint az objektumközpontúban. Ezért amikor az eseményszám nagy, az XES-dokumentum hosszabb, mint az OCEL.

3. Összegzés

A feladatom az volt, hogy összehasonlítsam a különböző eseménynapló-formátumokat abból a szempontból, hogy melyik a legalkalmasabb a folyamatbányász algoritmusokhoz, azaz melyik biztosítja az algoritmusok leghatékonyabb működését.

A szöveges formátumok (txt, csv, xml, json) nem szabványosak, ezért egységesítésük nehézkes, egyedi eljárásokat igényel. Az XES és OCEL szabványos formátumok, egyszerűbb az automatikus feldolgozásuk, de használatuk rendkívül terjedelmes naplóállományt eredményez.

A 2.5. szakaszban bemutatott számítások és grafikonok mutatják, hogy a legtömörebb leírás a szöveges, ahol egy eseményt egy sorban adunk meg. Ezek közül az SPMF-formátum az, amelyik kifejezetten folyamatbányászathoz van előkészítve, ezért a továbbiakban ezt a formátumot preferáljuk.

ESEMÉNYNAPLÓK A GYAKORLATBAN

MILEFF PÉTER

A mesterséges intelligencián alapuló informatikai rendszerek egyik kulcskérdése a megfelelő adathalmaz, hiszen ez teszi lehetővé a ráépülő folyamatok, a tanuló algoritmusok tervezését, a megfelelő betanulást. A magas minőségű adathalmaz tehát kulcsfontosságú, ezért kiemelt figyelemmel kell megközelíteni a problémát. A kutatás egyik alappillére tehát a megfelelő input-eseményhalmaz rendelkezésre álló formátumai közül az OCEL- és a CSV-formátumok gyakorlati szempontból való áttekintése. Jelen cikk az aktivitás naplók általános strukturális kérdéseit és azok Python környezetben való feldolgozhatóságát vizsgálja gyakorlati szempontból. Bemutatásra kerül egy olyan általános memóriabeli modellformátum, amely több típusú inputhalmaz általános kezelésére és leírására szolgál.

1. A kutatás célja és lépései

Napjaink egyre bonyolultabb társadalmi modellje egyre komplexebb ügyviteli folyamatokat eredményez. Ma már egy természetes és általános törekvés, hogy ezeket a folyamatokat informatikai rendszerekkel tudjuk megtámogatni, növelve ezzel a hatékonyságot. Ennek hiányában a folyamatok feldolgozása sokszor lassú lehet, a komplexitásuk miatt pedig számos esetben megfelelő támogatórendszer nélkül a támogatásuk és követésük nem kielégítő. Ma azonban nem elég csupán egy komplex ügyviteli rendszer megvalósítása, megfelelő automatizáltság szintén új követelményként jelent meg a piacon. Mivel az ügyviteli folyamatok komplexek, emiatt gyakran a kezelőfelület is meglehetősen sokrétű és bonyolult. Amennyiben egy-egy ilyen folyamatot részletesen kielemzünk, jól elkülöníthetők a folyamatmegoldási sémák. Meghatározhatók és felállíthatók olyan szabályok, amelyek a folyamatban szereplő döntések alapját képzik. Amikor egy szabálybázis már felállítható, gyakorlatilag az az a pont, amikor megkezdődhet a rendszer teljes, vagy akár részben történő automatizáltsága. Manapság már léteznek olyan rendszerek, ahol az ügyfél lényegében egy „automatával" beszél, vagy levelezik, ahol valamilyen mesterségesintelligencia-alapú döntéstámogató rendszer segít a probléma megoldásában, a folyamat lebonyolításában. Az ilyen rendszerek (RPA – Robotic Process Automation) segítségével a hatékonyság tovább növelhető.

Napjaink egyik új, RPA-alapú kutatási irányzata a szabályok automatikus felderítése. Az adminisztrációt végző rendszerek működésükből kifolyólag adatokkal és az ezeken transzformációkat végrehajtó folymatlépésekből tevődnek össze. Nagyon egyszerű példaként említhető egy árucikk online rendelésének a folyamata, ahol maga a rendelés egy egzakt, jól körülhatárolt folyamat. A folyamaton belül az egyes tevékenységek szempontjából lehetnek elágazások vagy alternatív útvonalak is. A fenti példánál maradva egy fizetés lebonyolítható online, vagy akár

utánvét útján is. A folyamat által használt adatok állapotváltozásaiból kinyerhető az a szabályminta, amelyet alkalmazva a folyamat akár teljes automatizáltsággal megoldható lenne. Ennek azonban az a követelménye, hogy az állapotok loggolva legyenek akár adatbázisban, akár egy vagy több naplófájlban. Mivel az RPA egy modern megközelítés a hatékonyság növelésére, így a legtöbb információs rendszer nincs megfelelően felkészítve arra, hogy olyan, az adatokon végzett változások megfelelő formában legyenek tárolva. A legtöbb esetben valamilyen logfájlok feldolgozásából indulunk ki, amelyek részletes és automatizált átvizsgálásával próbálunk szabályokat kinyerni.

Általános célként fogalmazható meg, hogy valamilyen információs rendszerből kinyert aktivitásnaplók alapján bizonyos események előre megjósolhatók legyenek valamilyen hatékonyan konfigurálható mesterséges neurális hálózatot alkalmazó modell segítségével. A mesterséges intelligencián alapuló informatikai rendszerek egyik kulcskérdése a megfelelő adathalmaz, hiszen ez teszi lehetővé a ráépülő folyamatok, a tanuló algoritmusok tervezését, a megfelelő betanulást. A magas minőségű adathalmaz tehát fontos, ezért kiemelt figyelemmel kell megközelíteni a problémát. A rögzített adatok, a munkafolyamat-lépések egy magasabb szintű szekvenciába, tranzakcióba szerveződnek. Mivel a rendszert egyszerre több felhasználó is használja, így egy olyan naplófájl fog létrejönni, amelyekben véletlenszerűen keverednek a különböző tranzakciókhoz tartozó tevékenységek.

Jelen kutatási munka több területen próbál előrehaladni. A hangsúlyt leginkább a támogatott formátumok valós, gyakorlati megvalósítására kell helyezni. A kutatás fő irányának a CSV formátum gyakorlati megvalósítását és az OCEL- (Object-Centric Event Logs) formátum alkalmazhatósági vizsgálatát tekinti.

A vizsgálat legfontosabb szempontjai:

- Aktivitásnaplók a gyakorlatban.
- CSV-formátum részletes analízise: köztudott, hogy a naplózás egyik klaszszikus és közkedvelt formátuma a CSV. A kutatás feladata az, hogy ennek a formátumnak a támogatása gyakorlati szinten is megvalósuljon.
- A CSV-formátum integrálása az első lépésként korábban már részben kidolgozott MLP modellben.
- OCEL-formátum támogatása: az OCEL egy 2021-ben kiadott szabványos formátum. Megvalósítása, Python alapú beépíthetőségének vizsgálata elengedhetetlen.
- A leíró szabványos formátum strukturális felépítésének megértése.
- A formátum gyakorlati szempontból történő elemzése.

- A formátum integrálhatósági kérdéseinek tisztázása. Rendelkezésre áll-e olyan Python library, amely már képes a formátum kezelésére.
- Fellelhető-e az interneten elegendő minta a formátumból, amely mind az implementációt, az adatvalidációt és az integrációt segítheti.
- Általános modellformátum: a különböző formátumok támogatása nem valósítható meg egy egységes, belső adatstruktúra nélkül. A kutatás ezt a fontos kérdéskört is körbejárja.
- Tesztek elvégzése különböző paraméterbeállítások mellett.
- Eredmények előzetes értékelése.

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

A felhasználóiaktivitás-monitorozás megvalósítása során két különböző típusú rendszert szokás megkülönböztetni gyakorlati szempontból. Vannak olyan rendszerek, amelyek már önmagukban tartalmazzák az eseménynaplózást, a fejlesztők és tervezők már a fejlesztés fázisában gondoltak az aktivitások naplózására. Ez sok esetben akár nagyon jól konfigurálható is. Például az ügyfélkapcsolat (CRM), az IT-szolgáltatáskezelés (ITSM), a rendeléskezelés és a munkafolyamat-rendszerek általában ebbe a kategóriába tartoznak. Az aktivitásfigyelés és folyamatbányászat szempontjából ezek a rendszerek a megvalósítási spektrum könnyebb oldalához tartoznak, hiszen az adatok nagy valószínűséggel megfelelő minőségben már rendelkezésre állnak naplók, adattáblák (előzménytáblák) és egyéb megoldások formájában, azok sokszor közvetlenül felhasználhatók.

A spektrum másik oldalán állnak azok a rendszerek, ahol nem állnak rendelkezésre a naplók kész formában, mert nem így lettek tervezve és kialakítva már a kezdetektől fogva. A felhasználói aktivitások figyelésére ezeknél gyakran valamilyen aktivitásfigyelő szoftvert alkalmazunk. Az aktivitásfigyelő szoftver rögzíti az alkalmazások és programok használatát a felügyelt munkaállomáson. A képernyőn megjelenő felhasználói tevékenységek egy előre kidolgozott és jól strukturált naplóba kerülnek. A naplók tehát információs adatbázisok, amelyek minden olyan tevékenységet tárolnak, amelyek aznap történtek. A mai modern technológiának köszönhetően számos lehetőség, megoldás áll rendelkezésre a monitorozásra, a tevékenységek figyelemmel kísérésére és kezelésére.

A gyakorlatban az aktivitásnaplók formátuma tetszőleges kialakítású lehet, a fő szempont mindig a logikus és egzakt felépítés az adatok tárolására és visszakereshetőségére. Ettől függetlenül három fő típus alakult ki az évek során:

- CSV: vállalati környezetben jól ismert és megszokott formátum
- XES: XML-alapú szabványos formátum az adatok tárolásához és továbbításához
- OCEL: egy 2021-ben kiadott szabványos objektumközpontú formátum

A továbbiakban a CSV- és OCEL-formátumokat tekintjük át részletesen és vizsgáljuk meg a kezelhetőségüket gyakorlati szempontból.

2.2. A CSV-formátum

A CSV-formátum kialakulását tekintve egy klasszikus formátum, amely egyike azon típusoknak, amelyek már a kezdetektől elérhetőek voltak. Vállalati szférában nagyon közkedvelt az Excelbe való közvetlen importálhatósága végett. A formátum számos előnnyel rendelkezik:

- Szöveges állomány: a szöveges formátumnak köszönhetően a naplóállomány bármely szerkesztővel megnyitható, tartalma átnézhető. Számos (akár) ingyenes programba (pl. LibreOffice) pedig jól importálható.
- Könnyű kezelhetőség: a formátum felépítése egyszerű, ezáltal egyedi feldolgozása hatékonyan megvalósítható. A fájl betöltése és adatainak oszlopokra és mezőkre bontása akár keretrendszerek nélkül is elvégezhető.
- Széles körű támogatottság: egyszerűsége révén számos információs rendszerbe hatékonyan integrálható.

Egy gyakorlati minta CSV-eseménynapló-formátumra:

```
CaseID,ActivityID,CompleteTimestamp
1,"open","2021-06-17 12:12:01"
1,"edit","2021-06-17 12:13:10"
1,"convert","2021-06-17 12:14:22"
1,"pack","2021-06-17 12:15:00"
1,"close","2021-06-17 12:15:30"
2,"open","2021-06-17 12:12:01"
2,"edit","2021-06-17 12:13:10"
2,"convert","2021-06-17 12:14:22"
2,"pack","2021-06-17 12:15:00"
2,"close","2021-06-17 12:15:30"
```

A fenti példa egy olyan CSV-formátumra mutat példát, amely egyszerű, felépítésében minimalizmusra törekedő. Az állomány minden egyes sora külön eseményt ír le, a bekövetkezésük sorrendjében. Az állomány fejléce tartalmazza az oszlopok neveit.

A mintában az oszlopok jelentése a következő:

- **CaseID:** a folyamat azonosítója, amelyhez az adott esemény tartozik. Gyakorlatilag egy számérték, amely tükrözi az információs rendszerben zajló folyamatot.
- **ActivityID:** az aktuális esemény neve, szöveges tartalom.
- **CompleteTimestamp:** az esemény bekövetkezésének pontos ideje. Fontos a másodpercalapú pontosság, különben az egyszerre párhuzamosan zajló események nem lesznek teljes mértékben megkülönböztethetők.

A fenti példában jól látszik, hogy két folyamat szerepel benne (caseid 1, caseid 2). Bár a folyamatok ebben a CSV-leírásban egymás után következnek, de a gyakorlatban a folyamatok lépései természetesen keveredhetnek egymással. Minden esetben az adatbeolvasó modul az, ami a napló tartalmát beolvasva a folyamatokhoz tartozó elemeket összegyűjti és rendszerezi.

2.2.1. A CSV-formátum hátrányai:

A bemutatott mint a CSV-fájl valójában már egy feldolgozásra előkészített és rögzített struktúrájú mintát mutat be. Míg az XES- és OCEL-formátumok már egy jól definiált struktúrába illeszkednek, addig a CSV-ről ez sajnos nem mondható el. Nagy problémát okoz az, hogy az oszlopok tetszőleges elnevezésűek lehetnek, valamint attól függően, hogy milyen rendszerből származnak, a oszlopok sorrendje is különbözhet. A különböző rendszerekből, akár rendszermodulokból érkező CSV-naplók felépítése tehát nagymértékben eltérhet, ezzel nehezítve a feldolgozást.

A hatékony adatfeldolgozás megvalósítása érdekében ilyenkor két lehetséges út áll előttünk:

- **Oszlopsorrend rögzítése:** ebben az esetben bármely alrendszerből vagy modulból érkezik az adat, mindegyik naplóban az oszlopok sorrendje előre rögzített. Bár triviálisnak tűnhet ez a megoldás, azonban nem mindenhol valósítható meg, az úgynevezett legacy rendszereknél nem minden esetben. Emellett pedig sokszor nem hatékony, mert ha valamilyen alrendszerben nincs olyan jellegű adat, akkor bekerül egy teljes üres oszlop a fájlba. A fájl esetenként tartalmazza az összes lehetséges attribútum oszlopot még akkor is, ha abban a modulban nincs is olyan (CSV-formátum-probléma).

- **Leírófájl alkalmazása:** A változó felépítés kezelésének egyik hatékony megközelítése ha minden naplófájlhoz egy leírófájl is tartozik. A leírófájl

definiálja a CSV-ben szereplő oszlopokat és azok pozícióját. Ez alapján a fájl feldolgozható.

Mivel az adatkinyerés során fontos, hogy a formátum struktúrája rögzítve legyen, ezért a fenti mintában látható struktúrát tekintjük a továbbiakban annak a minimális alapnak, amely egy rendszer megvalósításához szükséges. A minta kizárólag azt a minimális adatmennyiséget írja le, amely egy neurális hálózati modellen alapuló predikcióhoz szükséges. Nem szabad elfeledkezni azonban, hogy a kulcsadatokon kívül egyéb attribútumok feldolgozása is szükségessé válhat bizonyos esetekben. Ekkor azonban a leírófájl megléte elengedhetetlen.

2.2.2. CSV-formátum Python támogatottsága

Természetesen mivel a neurális hálózati modell elsődleges megvalósítási környezete a Python és a Keras, így magától adódik, hogy az adatok kezelését végző inputmodul megvalósítása is Python környezetben történjen. A Python 3.x több különböző lehetőséget kínál a CSV-betöltésre és -feldolgozásra. Jelen munka során választásunk a *Pandas* függvénykönyvtárra esett.

A *Pandas* egy nyílt forráskódú Python-csomag, amelyet széles körben használnak adattudományi/adatelemzési és gépi tanulási feladatokhoz. Egy másik, Numpy nevű csomagra épül, amely támogatja a többdimenziós tömböket. Az egyik legnépszerűbb adatmanipuláló csomagként a Pandas hatékonyan együtt tud működni több más adattudományi modullal a Python-ökoszisztémán belül, és jellemzően minden Python-disztribúcióban megtalálható.

CSV betöltése Pandas-csomag segítségével:

```
import pandas as pd
df = pd.read_csv (r'Path where the CSV file is stored\File name.csv')
print (df)
```

Jól látható, hogy maga a CSV-betöltés a Pandas-csomag megfelelő telepítése után, viszonylag könnyen elvégezhető. Természetesen számos paraméterezési lehetősége kínál a Pandas. Példa CSV-betöltésre, csak bizonyos oszlopokra:

```
import pandas as pd
data = pd.read_csv (r'C:\Users\Ron\Desktop\Clients.csv')
df = pd.DataFrame(data, columns= ['Person Name','Country'])
print (df)
```

A fenti példák jól mutatják, hogy a Pandas CSV-megoldása jó alapot kínál egy saját tervezésű adatmodul számára. A cikk a későbbiekben erre is kitér.

2.3. OCEL formátum

A kutatás során az általános adatmodul tervezése központi szerepet töltött be. A CSV-formátum mellett egy új, szabványos formátum, az OCEL (Object-Centric Event Logs) (http://www.ocel-standard.org/) vizsgálata került a középpontba Az OCEL egy 2021-ben létrejött formátum, amelynek célja, hogy egy általános ajánlást biztosítson az objektumalapú eseményadatok tárolására és hordozására. Kiegészíti az XES-szabványt, az eseményadatok cseréjére szolgáló hivatalos IEEE-t, és a legtöbb folyamatbányászati eszköz támogatja. Az OCEL nem ír elő esetfogalmat, ezért a klasszikus eseménynaplók (például XES-formátumban) és a valós információs rendszerek (SAP, Oracle, Inform, Salesforce, MS Dynamics stb.) adatok között helyezkednek el. Ezért az OCEL használható a folyamat holisztikusabb áttekintésére, azaz különböző típusú objektumok (pl. megrendelések, cikkek, ügyfelek, fizetések és szállítmányok) tárolhatók egyetlen nézőpont érvényesítése nélkül. Az OCEL azonban köztes tárolási formátumként is használható például az SAP és a klasszikus folyamatbányászati technikák között.

A szabvány két fájlformátumtípust támogat: JSON-OCEL, XML-OCEL. Az alábbi ábrák egy-egy mintát mutatnak be ezekről a típusokról:

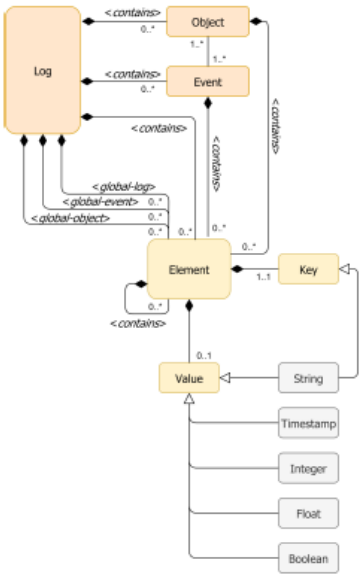
```
<events>
  <event>
    <string key="id" value="e1"/>
    <string key="activity" value="place_order"/>
    <date key="timestamp" value="2020-07-09T08:20:01.527+01:00"/>
    <list key="omap">
      <string key="object-id" value="i1"/>
      <string key="object-id" value="o1"/>
      <string key="object-id" value="i2"/>
    </list>
    <list key="vmap">
      <string key="resource" value="Alessandro"/>
      <float key="prepaid-amount" value="200.0"/>
    </list>
  </event>
  <event>
    <string key="id" value="e2"/>
    <string key="activity" value="check_availability"/>
    <date key="timestamp" value="2020-07-09T08:21:01.527+01:00"/>
```

1. ábra. *XML-alapú OCEL-eseménynapló-minta*

```
"ocel:events": {
  "e1": {
    "ocel:activity": "place_order",
    "ocel:timestamp": "2020-07-09T08:20:01.527+01:00",
    "ocel:omap": [
      "i1",
      "o1",
      "i2"
    ],
    "ocel:vmap": {
      "resource": "Alessandro",
      "prepaid-amount": 200.0
    }
  },
  "e2": {
    "ocel:activity": "check_availability",
    "ocel:timestamp": "2020-07-09T08:21:01.527+01:00",
```

2. ábra. *JSON-alapú OCEL-eseménynapló-minta*

Az OCEL-szabvány jelenleg az 1.0 verziónál tart. Ezen verziójú formátum elemeit és azok kapcsolatát mutatja be az alábbi ábra:



3. ábra. *OCEL-eseménynapló metamodel felépítése*

Az OCEL-támogatás beépítésének mérlegelése mellett szól, hogy objektum-alapú formátum, amely megfelelő szinten struktúrált, mindamellett pedig modern. Az OCEL-formátum támogatása jelenleg a Python 3 verziótól kezdve érhető el, azonban jelenleg nagyon kezdetleges. A hivatalos Python tárolókban az **ocel-standard** csomag biztosítja a feldolgozást, amely jelenleg a 0.0.3.1 verziószámon elérhető. Bár a csomag nagyon kezdetleges, már elegendő lehetőséget adott arra, hogy a feldolgozás elvégezhető legyen.

A kidolgozott, jelenleg egyszerű OCEL-formátumot betöltő és minimálisan feldolgozó mintakód a következő:

```
import ocel;

log_object = ocel.import_log("minimal.jsonocel")
types = ocel.get_object_types(log_object)
global_event = ocel.get_global_event(log_object)
objects = ocel.get_objects(log_object)
events = ocel.get_events(log_object)

activity_list = []
omap_list = []

for kk in events.keys():
    event = events[kk]
    activity = event['ocel:activity']

    # get all activities
    if (activity not in activity_list):
        activity_list.append(activity)

    omap_content = event['ocel:omap']

    # get all omap objects
    for oo in omap_content:
        if (oo not in omap_list):
            omap_list.append(oo)

# make a dictionary with numbers and activity names
activity_dict = dict()
i = 0
for ii in activity_list:
    activity_dict[ii] = i;
    i+=1

print(activity_dict)
```

Jelen mintakód arra mutat példát, hogy hogyan lehet azokat az alapinformációkat (*types, global_event, objects, events*) kigyűjteni a betöltés után, amelyeket képesek vagyunk használni, amelyre már egy bonyolultabb rendszer is alapozható. Bár a Python-támogatás jelenleg még kezdetleges, de már így is megvalósítható az OCEL-formátum beépítése a rendelkezésre álló alaprutinok segítségével a korábbi CSV- és XES-formátumok mellett.

3. Általános leíró formátum bevezetése

A projekt során megvalósítandó rendszer tervezésekor célszerű már az elején arra is gondolni, hogy az adatokat biztosító réteg akár több irányból érkező adatokat is ki tudjon szolgálni. A bemeneti oldal nem korlátozható le egyetlen formátumra. Szoftvertechnológiai megvalósítási szempontból több formátum kezelésének problémája számos más területen is megjelenik. Célszerű megvalósítását az alábbiakban részletezzük.

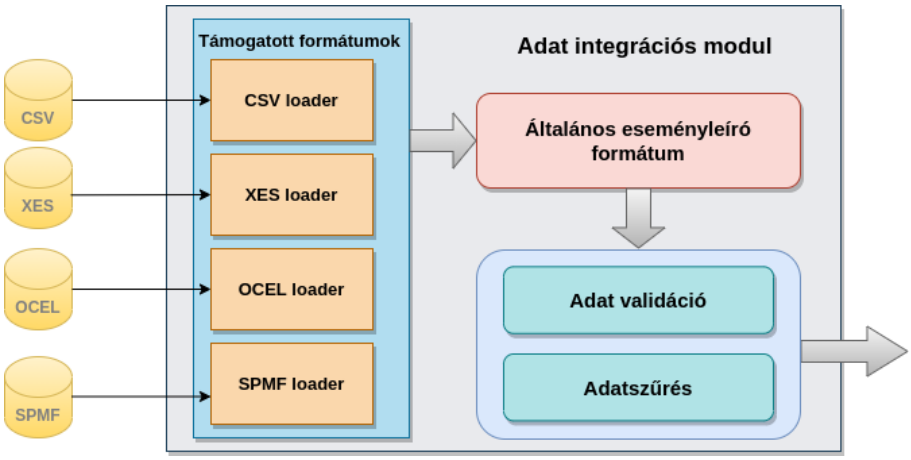
Alapvető probléma, hogy a különböző formátumok támogatása különböző, formátum-specifikus megvalósítást kíván meg a rendszertől. Fontos szempont, hogy külön kell választani az adatokat szolgáltató réteget, a predikciót végző logikai modultól. Bár elméletben megvalósítható az az irány, miszerint a neurális hálózaton alapuló logikai modul minden formátumot megvalósítson, képes legyen a formátumspecifikus adatokat értelmezni, a gyakorlatban azonban ez téves tervezési minta. Ugyanis ebben az esetben az adatintegrációs logika egy olyan modulba van integrálva, amelynek nem ez a fő szerepe. A megvalósítási szinten ilyenkor kényszerítetten duplikált, vagy nagyon hasonló metódusok jelennek meg, a kód nem tiszta és jól karbantartható. A hosszú távú fejlesztés nem hatékony.

A megoldás számos előnnyel rendelkezik:

- Ezzel a megoldással lehetővé válik, hogy a különböző forrásokból származó adatok egy adott struktúrában jelenjenek meg.
- Az adatok validációja és transzformációja egy helyen elvégezhető.
- Bármely olyan modul, amelynek adatra van szüksége, már egy egységes modellt lát, nem pedig különböző típusú naplóstruktúrákat.

A kidolgozott általános logikai modell lehetővé teszi, hogy a későbbiekben akár tetszőleges formátum integrálható legyen. A *4. ábra* ezt logikailag mutatja be.

Az adatintegrációs modul végeredményképpen tehát egy validált adathalmazt képes átadni a további modulok számára. Gyakorlatilag tetszőleges leíró formátum megvalósítható ezzel a megközelítéssel. Az adatok kezelése pedig a betöltés után egységes logika alapján transzformálható és használható fel.



4. ábra. *Adatintegrációs modul logikai felépítése*

Egyéb elvárt funkciók:

Eseményablakok támogatása:

- bemenetként kapott eseményszekvencia-listák átalakítása,
- egyforma nagyságú ablakokra darabolása.

Eseménysorok kiegészítése:

- az eredeti, hiányos adatok pótlása; tipikus művelet az eseménykezdet, és -vég beillesztése az esemény id sorba.

Eseménysorok átalakítása:

- a feldolgozás során bizonyos értékek módosítása, mappelése, lambda kifejezések végrehajtása stb.

Eseményszekvenciák logikai ellenőrzése:

- a szintaktikailag megfelelő adathalmaz logikai ellenőrzése.

3.1. Általános leíróformátum-struktúra

Jelen munka során az közös eseménymodell-struktúra logikai felépítésének kidolgozásakor az első szempont az egyszerűség volt. Ez alapján a javasolt struktúra a következő:

Log: a legmagasabb logikai szint, amely egy eseménynaplófájlt reprezentál. A Log trace-ek halmaza.

Trace: egy adott folyamat/ügy megvalósulása. Minden trace egyedi azonosítóval rendelkezik (trace_id) és eseményekből (Event) tevődik össze.

Event: A legalacsonyabb szintű struktúra. Egy elemi eseményt tárol. Minimálisan szükséges adatok: név, idő.

Az eseménymodell-struktúra definiálását Python nyelven végeztük el:

```
#
# Represents a simple event/activity
#
class Event:

    def __init__(self, name: str, timestamp: str):
        self.name = name
        self.timestamp = timestamp

#
# Represents a simple Case/Trace
#
class Trace:

    def __init__(self, trace_id, event_list=None):
        self.trace_id = trace_id

        if event_list is None:
            event_list = []

        self.event_list = event_list

    def add_event(self, event: Event):
        self.event_list.append(event)

    def print_traces(self):
        for event in self.event_list:
            logging.info(str(self.trace_id) + " - " + event.name)

    def get_number_of_events(self):
```

```
        return len(self.event_list)

#
# Represents a log file
#
class Log:

    def __init__(self, name: str, trace_list=None):
        self.name = name

        if trace_list is None:
            trace_list = []

        self.trace_list = trace_list

    def add_trace_list(self, _trace_list):
        self.trace_list.append(_trace_list)

    def add_trace(self, trace):
        self.trace_list.append(trace)

    def print_log_traces(self):
        for trace in self.trace_list:
            logging.info(trace.trace_id)
```

A struktúra jelenleg csak a legfontosabb adatokat képes tárolni. Amennyiben nem tudná a jövőben kiszolgálni az igényeket, úgy tetszőleges szinten bővítésre kerülhet.

3.2. CSV-betöltés gyakorlati megvalósítása

Az alábbi kód bemutatja az elkészült CSV-olvasót, amely már a fent definiált általános eseménynapló-struktúrába alakítja a bemeneti adatokat.

```
def load_trace_list_csv(file_name):
    # CaseID,ActivityID,CompleteTimestamp
    dataset_all = pandas.read_csv(file_name)

    # This will be the case array
    trace_array = []
```

```
    # Parse CSV data
    for index, row in dataset_all.iterrows():

        activity = row['ActivityID']
        case_id = row['CaseID']
        timestamp = row['CompleteTimestamp']

        found_case = False
        for oo in trace_array:
            if oo.trace_id == case_id:
                new_event = Event(activity, timestamp)
                oo.add_event(new_event)
                found_case = True
                break

        if not found_case:
            # create new event
            new_event = Event(activity, timestamp);
            # create new trace
            new_trace = Trace(case_id)
            # add event to trace
            new_trace.add_event(new_event)
            # add trace
            trace_array.append(new_trace)

    log = Log(file_name)
    log.add_trace_list(trace_array)

    for oo in trace_array:
        oo.print_traces()

    return log
```

3.3. Automatizált tesztelés Python-környezetben

Tesztelés az alapja a SOLID szoftver fejlesztésének. Többféle tesztelési típus létezik, de a legfontosabb típusa a unit tesztelés. A tesztesetekkel képesek leszünk prezentálni azt, amit manuálisan is megtennénk viszont az emberből adódóan képesek vagyunk hibázni és egy komplex rendszernél funkciókat kifelejteni, ezáltal emberi hibából adódóan nem kerül leteszteléstre minden funkció. Automatizált

tesztelés továbbá időt is képes megspórolni, hisz előre le vannak fektetve a tesztesetek és csak a kiértékelésnél kell vizsgálni, mely tesztesetek feleltek meg és melyek nem.

A Python standard könyvtára tartalmazza a unittest modult. Ez egy *TestCase* osztályt biztosít a fejlesztőknek, amelyből saját osztályunkat származtathatjuk. A projekt kapcsán, a jelenlegi fázis alapján különböző fájlformátumra, valamint az általános modellformátumra célszerű teszteket definiálni.

A unittest modul különféle eszközöket biztosít a tesztek csoportosítására és programozott futtatására (Tesztek betöltése és futtatása). De a legkönnyebb mód a tesztek feltérképezése (discovery). Ezt az opciót csak Python 2.7-ben vezették be. 2.7 előtt a nose modult használhatjuk a feltérképezéshez és a tesztek futtatásához. A nose modulnak néhány más előnye is van, mint pl. tesztfüggvények futtatása, a tesztesetekhez szükséges osztály létrehozása nélkül.

Unit teszt alapú tesztek feltérképezése és futtatása:

```
> python -m unittest discover
```

Az unit teszt parancs végignézi az összes file-t és alkönyvtárat, lefuttatja az összes tesztet, amit talál, és egy riportot ad vissza a futása során. Ha szeretnénk látni, melyik teszteket futtatja, add hozzá a -v kapcsolót:

```
> python -m unittest discover -v
```

3.3.1. Tesztelés a gyakorlatban

A támogatott eseménynapló-formátumok betöltését Python-környezetben fejlesztett programokkal végeztük el. A fejlesztési környezet Linux-platform volt.

Jelenleg két darab központi unit teszt fájl került létrehozásra. Az egyik a fájlok betöltését, a másik pedig a feldolgozást hivatott validálni. Az alábbi példa, a fájlok helyes betöltését validálja. Terjedelmi okokból csak néhány kisebb részlet kerül bemutatásra.

```
import unittest
import logging
import sys

from util.file import load_trace_list_xes
from util.model import Event, Trace, Log
from util.file import load_trace_list_csv
```

```
def setup_logger():
    logging.basicConfig(filename='unittest.log', level=logging.INFO)
    stdout_handler = logging.StreamHandler(sys.stdout)
    a_logger = logging.getLogger()
    a_logger.addHandler(stdout_handler)

class Test(unittest.TestCase):
    def test_load_trace_list_xes(self):
        setup_logger()

        # GIVEN
        file_name = 'xes_samples/simple_abc_bd.xes'

        # WHEN
        output = load_trace_list_xes(file_name)

        # THEN
        self.assertTrue(isinstance(output, Log))
        self.assertEqual(len(output.trace_list), 3)
        self.assertTrue(isinstance(output.trace_list[0], Trace))
        self.assertEqual(len(output.trace_list[0].event_list), 3)
        self.assertTrue(isinstance(output.trace_list[0].event_list[0], Event))

    def test_load_trace_list_csv(self):
        setup_logger()

        # GIVEN
        file_name = 'csv_samples/basic_event.csv'

        # WHEN
        log = load_trace_list_csv(file_name)

        # THEN
        self.assertEqual(len(log.trace_list), 1)

        trace = log.trace_list[0]
        self.assertEqual(trace[0].get_number_of_events(), 5)

if __name__ == '__main__':
    unittest.main()
```

A tervezéskor célunk az volt, hogy a unit tesztelés eredménye megmaradjon. Ezért integráltuk a Python logger funkcióját. A fenti kódból jól látszik, hogy a validáció eredményét egy unittest.log fájlba teszi bele.

A különböző formátumok tesztelésére alapmintafájlokat hoztunk létre, a tesztelés ezeken történik meg.

A betöltött adatok feldolgozása szintén unit tesztekkel validált hasonlóan az előzőhöz.

Részlet a tesztelést elvégző fájlból:

```
def test_create_window_x_y(self):
        # GIVEN
        window_size = 3
        trace_sequence_list = [['a', 'b', 'c', 'd', 'e', 'f', 'g'], ['e', 'h', 't', 'u']]

        # WHEN
        x, y, all_event = create_window_x_y(trace_sequence_list, window_size)

        # THEN
        self.assertEqual(
                    [[0, 0, '^'],
                     [0, '^', 'a'],
                     ['^', 'a', 'b'],
                     ['a', 'b', 'c'],
                     ['b', 'c', 'd'],
                     ['c', 'd', 'e'],
                     ['d', 'e', 'f'],
                     ['e', 'f', 'g'],

                     [0, 0, '^'],
                     [0, '^', 'e'],
                     ['^', 'e', 'h'],
                     ['e', 'h', 't'],
                     ['h', 't', 'u']],
                    x
                    )
        self.assertEqual(['a', 'b', 'c', 'd', 'e', 'f', 'g', end_sign, 'e', 'h', 't', 'u', end_sign], y)
        self.assertEqual(set([start_sign, 'f', 't', end_sign, 'u', 'a', 'd', 'b', 'e', 'c', 'g', 'h']),
set(all_event))

    def test_create_window_x_y_custom_signs(self):
```

```
    # GIVEN
    window_size = 3
    trace_sequence_list = [['a', 'b', 'c', 'd']]
    util.preprocessing.start_sign = 'START'
    util.preprocessing.end_sign = 'END'

    # WHEN
    x, y, all_event = create_window_x_y(trace_sequence_list, window_size)

    # THEN
    self.assertEqual(
        [[0, 0, 'START'],
         [0, 'START', 'a'],
         ['START', 'a', 'b'],
         ['a', 'b', 'c'],
         ['b', 'c', 'd']],
        x
    )
    self.assertEqual(['a', 'b', 'c', 'd', 'END'], y)
    self.assertEqual(set(['START', 'END', 'a', 'd', 'b', 'c']), set(all_event))
```

4. Összegzés

A felhasználói események naplózása ma már kulcsfontosságú a modern információs rendszerekben. A rendelkezésre álló események alapján kidolgozhatók olyan mesterségesintelligencia-modellek, amelyek segítségével az ügyviteli folyamatok részben automatizálhatók, valamint akár a szűk keresztmetszetek felderíthetők. A felhasználói események naplózásának a gyakorlatban számtalan megvalósulási formája lehet. Jelen munkában két formátumot vizsgáltunk meg, azok gyakorlati megvalósíthatóságát Python-környezetben teszteltük. Egy informatikai rendszerben több formátum támogatásának hatékony megoldására pedig egy általános, memóriabeli leíró struktúrát javasoltunk, amely segítségével egységesített az adatmodul outputja.

A kutatás további lehetőségeként célszerű lehet további formátumok integrálhatóságának megvizsgálása, egy egységes validációs eljárás kidolgozása, amely részben fájlformátum-specifikus, részben pedig már az általános memóriabeli formátumon hajtódik végre. Valamint célszerű további attribútumokkal való bővíthetőségek vizsgálata.

FOLYAMATLEÍRÓ MODELLEK ÉS AUTOMATIKUS FELTÁRÁSUK

BAKSÁNÉ VARGA ERIKA

Egy vállalat működését az üzleti folyamatai határozzák meg. Ezek a munkafolyamatok olyan feladatok sorozatából állnak, amelyeket az alkalmazottak rendszeresen végeznek konkrét célok elérése érdekében. Ezeknek a folyamatoknak a formális leírása kulcsfontosságú tényező a vállalat működésének hatékonyabbá tétele szempontjából. A számítógépes munkavégzés során elvégzett tevékenységeket (bekövetkezett eseményeket) eseménynaplók tárolják. Mivel az eseménynaplót a rendszer automatikusan rögzíti, első körben feltételezzük, hogy valós adatokat tartalmaz. Jelen kutatás célja vizsgálni azokat a módszereket, amelyekkel a napló adataiból kikövetkeztethető az az általános folyamatmodell, amelyre a napló összes eseményszekvenciája illeszkedik. A vizsgálatot azért végezzük, hogy kiválasszunk benchmark eljárásokat a saját módszerünk értékeléséhez.

1. Bevezetés

A kutatás első lépéseként áttekintem a leggyakrabban alkalmazott folyamatleíró nyelveket és azokat az algoritmusokat, amelyek a számítógépes munkavégzés során rögzített eseménynaplókból ilyen modelleket képesek automatikusan előállítani.

A grafikus üzleti folyamatmodellekben (Business Process Modeling, BPM) az a közös, hogy a folyamatokat tevékenységek sorozataként írják le, ahol a tevékenységek sorrendje egymásra épülést, időbeli függőséget jelöl. Ezt egy gráfban ábrázolják, amely csomópontokból és a csomópontok közötti irányított élekből épül fel, jelezve a folyamatok kronológiai sorrendjét. Csoportosításuk szerint vannak

- hagyományos folyamatmodellező nyelvek (pl. Petri háló, IDEF, EPC, RAD),
- munkafolyamat-modellező nyelvek (pl. WPDL, BPMN), és
- objektumorientált nyelvek (pl. UML).

A legrégebbi és legelterjedtebb modell a Petri-háló, amelynek létezik néhány magasabb szintű kiterjesztése, míg a legkifejezőbb nyelv a BPMN. Ezekhez a modellekhez szabványos XML-alapú adatcsere-formátumokat is kidolgoztak. A BPMN-szabvány magában foglalja az XPDL nyelvet (XML Process Definition Language), a Petri hálók pedig automatikusan feldolgozhatók a PNML nyelv (Petri Net Markup Language) segítségével.

A Workflow Patterns Initiative szisztematikus elemzést végzett, hogy milyen jellemző szerkezeteket használnak a BPM-nyelvek. Ennek eredményeként létrehoztak egy sablon gyűjteményt, és kidolgozták a YAWL nyelvet, amellyel minden azonosított minta leírható. Ezek a minták a munkafolyamatot többféle nézőpontból írják le, például vannak vezérlési minták, adatminták vagy erőforrásminták.

Vizsgálataink során mi a vezérlő szerkezetekre összpontosítunk. Ebben a csoportban 43 mintázat található 8 osztályba sorolva, melyek közül a legfontosabbak:

- szekvenciális végrehajtása a tevékenységeknek,
- XOR, azaz kizárólagos választás több tevékenység közül, amelynek be kell fejeződnie a következő tevékenység indítása előtt,
- OR, azaz egy vagy több tevékenység kiválasztása az opciók közül. Az összes kiválasztott tevékenységnek be kell fejeződnie ahhoz, hogy a következő tevékenység indulhasson.
- AND, azaz az összes tevékenység párhuzamos indítása, majd ezek szinkronizálása.

A fent említett modellek speciális jelölést alkalmaznak a vezérlés menetét befolyásoló elemekre. Ezek az elemek összekapcsolják a tevékenységeket a grafikus folyamatmodellgráfban, de az eseménynaplóban – ami a folyamatmodell egy konkrét, megvalósult példánya – nincs jelük. Ezért ezek feltárása külön módszereket igényel.

2. Kutatási eredmények összesítése

Az egyik legszélesebb körben tanulmányozott folyamatbányászati művelet az automatikus folyamatfeltárás. A folyamatfeltáró módszerek bemenete egy eseménynapló, kimenetként pedig egy üzleti folyamatmodellt állítanak elő, ami az eseménynaplóban rögzített vagy az alapján feltételezett tevékenységek közötti kapcsolatokat ábrázolja a vezérlésfolyam szempontjából. Az eljárás során előállított modellnek az alábbi elvárásokat kell teljesítenie:

- az eseménynaplóban található összes esetet lefedi (maximálisan illeszkedik a bemenetre),
- nem generál olyan eseteket, amelyekhez hasonlót az eseménynapló nem tartalmaz (pontos),
- minden olyan eset előállítható belőle, ami az eseménynaplóban található esetekhez hasonló (általános), és
- a lehető legegyszerűbb (a komplexitása minimális).

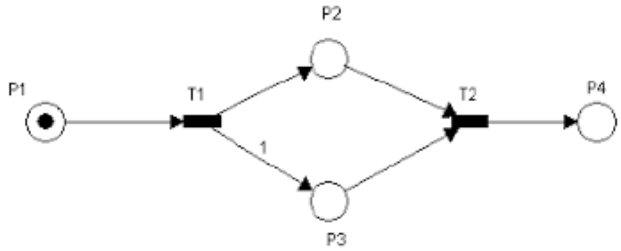
Vizsgált folyamatmodellező nyelvek:

Petri-háló. A legelterjedtebb folyamatmodellezési eszköz. A modellben a csomópontoknak két típusa van:

- normálállapot-hely (place),
- állapotátmenet (transition).

Kapcsolat csak normál állapot és állapot átmenet csomópontokat köthet össze. Az átmeneteknél így beszélhetünk bemenő és kimenő helyekről.

A normál állapotok és átmenetek mellett státuszjelölő elemek (tokenek) is vannak a modellben. A tokenek a létezésükkel jelzik a kapcsolódó állapot teljesülését. A modellben sajátos feltétel kell az átmenet megvalósulásához. Egy átmenet csak akkor történik meg (tüzelés), ha minden bemenő helyen van token. Az átmenet megvalósulása során a tokenek átkerülnek a kimenő helyekre. Itt nem feltétlenül érvényesül a tokenmegmaradás törvénye. Az élekhez rendelhetünk kapacitásértéket is, mely megadja az átmenet során érintett tokenek darabszámát.



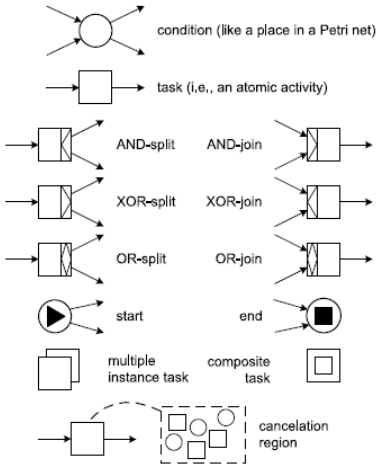
Az alap Petri-hálóhoz többféle továbbfejlesztést dolgoztak ki, amelyek további szempontokkal bővítik ki az alapmodellt. A fontosabb bővítési irányok:

- színezett Petri-háló, amikor a tokenek további jellemzőket is tartalmazhatnak,
- hierarchikus Petri-háló,
- időbeliséget, ütemezést tartalmazó Petri-háló.

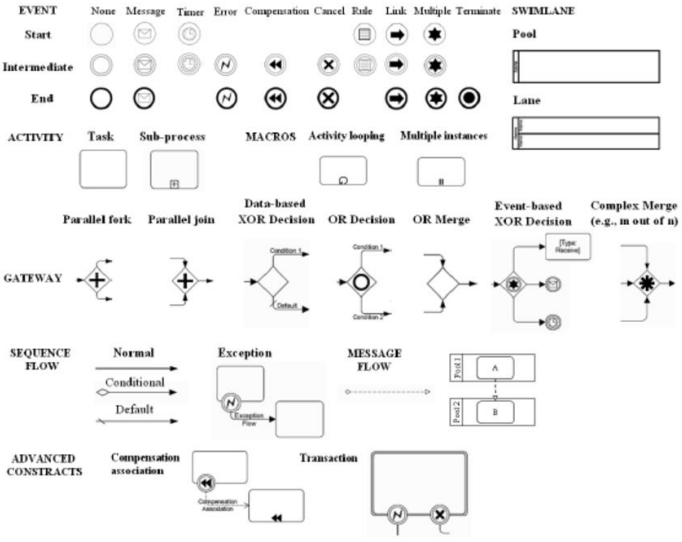
A Petri-háló speciális esete a Workflow háló, amelyben az alábbi két megkötés él:

- Egyetlen induló (start) és egyetlen cél (záró) csomópont létezik.
- Minden csomópont rajta van egy start-cél útvonalon.

YAWL (Yet Another Workflow Language). Workflow szemléletű modellezési eszköz. A nyelv több tipikus workflow mintát biztosít építőkőként. A minták több különböző szempontra is kiterjednek, így vannak többek között vezérlésielem-minták, adatstruktúra-minták, műveleti minták és hibajelenség-minták. A nyelv hatékonyságát ezen minták biztosítják, hiszen a mintákat felhasználva gyorsabban lehet komplex rendszereket felépíteni. A modell vezérlő elemei az alábbi ábrán láthatók.

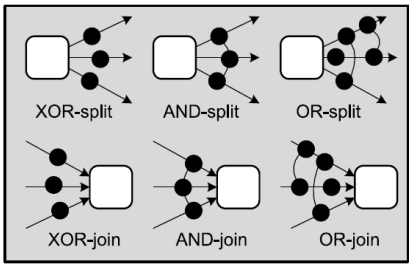


BPMN (Business Process Model Notation). A YAWL formalizmushoz hasonló modellezési nyelv, mely az iparban igen széles körben elterjedt, elsősorban a funkcionális gazdagsága miatt. Jelölésrendszerét az alábbi ábra szemlélteti.

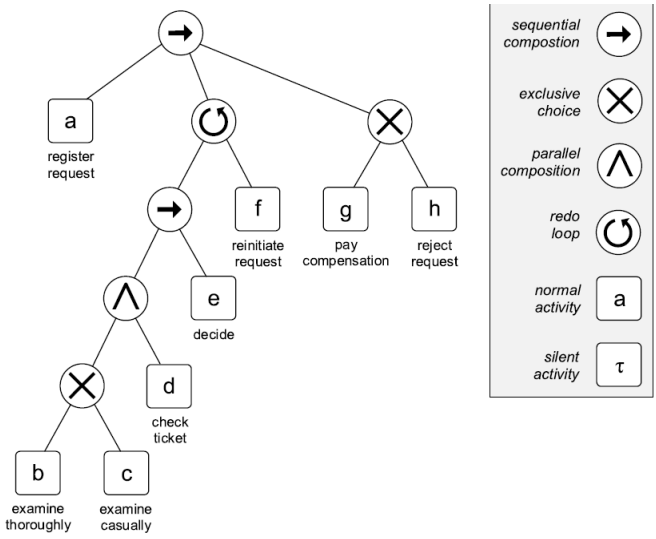


Causal Nets. Ez a modell a folyamatot olyan gráfként reprezentálja, ahol a csomópontok az eseménynaplóban is megjelenő tevékenységek és az élek a tevékenységek

közötti okozati összefüggést képviselik. A modellben a tevékenységekhez hozzárendelik a bejövő függőségek és a kimenő függőségek halmazát. Az okozati kapcsolatoknál azonban összetett összefüggések is megjelenhetnek, ezért a modellben új kapcsolati formák is megjelennek, melyek a szokásos konjunkció és diszjunkció operátorokon alapulnak.



Folyamatfa. A folyamatok leírásánál az általános gráfalapú módszerek nem feltétlenül adnak érvényes leírást, nem tudják igazán kontrollálni a magasabb szinten megjelenő formákat. A folyamatfa-alapú leírás ezzel szemben egy strukturális, tartalmazásalapú hierarchikusmodell-leírás, ami egy kevésbé rugalmas, de ellenőrzöttebb modellt eredményez. A fában a belső csomópontok strukturális egységek, míg a levelek elemi műveleteket jelölnek. Az alábbi ábrán látható folyamatfa Wil van der Aalst professzor *Process Mining* című könyvében található.



2.1. Elvégzett kísérletek bemutatása

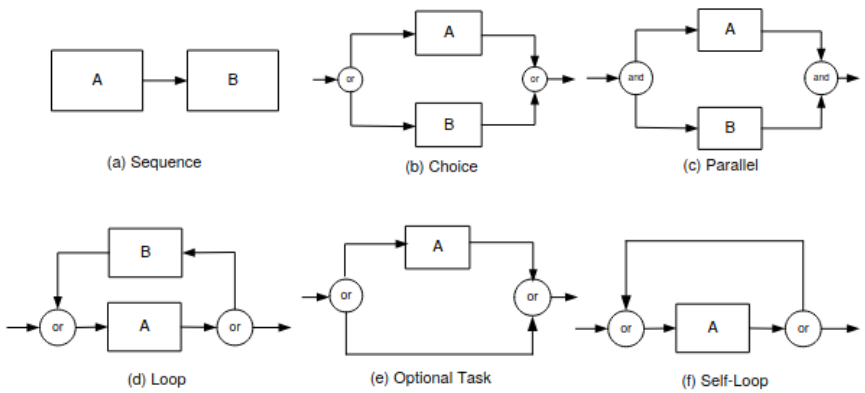
A vizsgálat alá vont eljárások kiválasztásának szempontjai:
1) 10 évnél nem régebbi módszer,
2) valós eseménynaplón tesztelt eljárás,
3) a feltárt folyamatmodell procedurális típusú, azaz az események végrehajtási sorrendjét (control flow) határozza meg.

Ez alapján 23 folyamatfeltáró eljárást vizsgáltam meg 2 szempont szerint:
1) a feltárt modell leíró nyelve (pl. BPMN, Petri-háló stb.), és
2) a feltárt vezérlőelemek (AND, OR, XOR).

2.2. Kiértékeléshez használt referencia folyamatok és értékek bemutatása

Az eseménynaplókból procedurális folyamatmodellt feltáró létező eljárások:
HK, Inductive Miner, Process Skeletonization, Evolutionary Tree Miner, Aim, Competition Miner, Directed Acyclic Graphs, CN Mining, alpha$, Maximal Pattern Mining, DGEM, RegPFA, BPMN Miner, CSM Miner, TAU Miner, PGminer, ProM-D, Proximity Miner, Heuristics Miner, Split miner, Fodina, Stage miner, Decomposed Process Miner.

Ezeknek az eljárásoknak a vizsgálata során azt tartottam szem előtt, hogy a leggyakoribb folyamatvezérlési minták közül, melyeket képesek feltárni:



2.3. Kiértékelések eredményeinek bemutatása

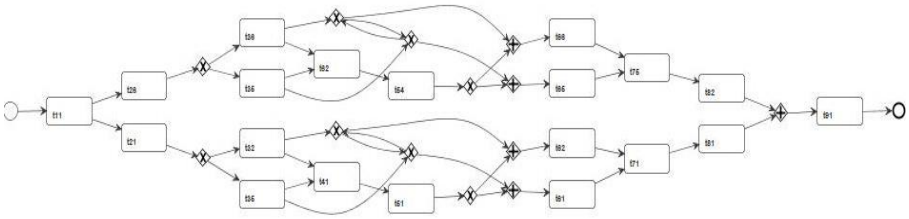
A vizsgálati eredményeket az alábbi táblázatban foglalom össze.

Eljárás neve	A feltárt modell leíró nyelve	Feltárt vezérlőelemek			
		AND	XOR	OR	Ciklus
Aim	Petri-háló	+	+		+
alpha$	Petri-háló	+	+		+
BPMN Miner	**BPMN**	**+**	**+**	**+**	**+**
CN Mining	Causal net (ok-okozati háló)	+	+		+
Competition Miner	BPMN	+	+		+
CSM Miner	Állapotgép	+	+		+
Decomposed Process Miner	**Petri-háló**	**+**	**+**	**+**	**+**
DGEM	BPMN	+	+		+
Directed Acyclic Graphs	Irányított gráf		+		
Evolutionary Tree Miner	**Folyamatleíró fa**	**+**	**+**	**+**	**+**
Fodina	BPMN	+	+		+
Heuristics Miner	BPMN	+	+		+
HK	Petri-háló	+	+		+
Inductive Miner	**Folyamatleíró fa**	**+**	**+**	**+**	**+**
Maximal Pattern Mining	Causal net (ok-okozati háló)	+	+		+
PG miner	Részben rendezett gráf	+	+		
Process Skeletonization	Irányított gráf		+		+
ProM-D	Petri-háló	+	+		+
Proximity Miner	Causal net (ok-okozati háló)	+	+		+
RegPFA	Petri-háló	+	+		+
Split miner	BPMN	+	+		+
Stage miner	Causal net (ok-okozati háló)	+	+		+
TAU Miner	Petri-háló	+	+		+

2.4. Eredményeket szemléltető képernyőképek

A vizsgált folyamatmodell-leíró formalizmusok grafikus nyelvek. A folyamatfeltáró algoritmusok ezek közül csak a folyamatleíró fákat és a gráfokat képesek előállítani. A BPMN és Petri-háló modellek csak konverzió után vizualizálhatók. Ezek esetében a modellt feltáró eljárások kimenete egy számítógéppel olvasható, XML-alapú formátum. A BPMN esetén egy XPDL- (XML-based Process Definition Language), Petri-háló esetén pedig egy PNML- (Petri Net Markup Language) fájl.

A PNML folyamatmodellfájlokat a ProM szoftverrel lehet megjeleníteni (www.promtools.org). A PNML egy XML-alapú, számítógéppel olvasható (machine readable) formátum. A ProM szoftverben BPMN grafikus formátumra konvertáltam, majd exportáltam jpg, png, pdf, eps stb. formátumba és az alábbi eredményt kaptam:



A BPMN-gráfban a téglalapok az események (tevékenységek) és a rombusszal jelölt elemek az átjárók (gateway). Az x-szel jelölt átjáró azt jelenti, hogy az adott ponton szétválik a folyamat vezérlése egymást kizáró ágakra (exclusive gateway), azaz a folyamatpéldányokban csak az egyik útvonal valósul meg az ebből a pontból kiinduló ágak közül (OR-split). A döntés egy megadott feltétel kiértékelésétől függ. A +-szal jelölt átjáró az a pont, ahonnan akkor lehet továbblépni, ha a megelőző ágak közül az aktív befejeződött (OR-join). Ezek az átjárók a modell feltárása során kerülnek meghatározásra, a naplóállományokban nincs nyomuk.

3. Összegzés

A vizsgálat során áttekintettem a legelterjedtebb folyamatmodellező nyelveket, és az automatikus feltárásukat megvalósító módszereket azzal a céllal, hogy kiválasszam azokat az eljárásokat, amelyekkel össze fogjuk hasonlítani a saját algoritmusunk teljesítményét.

A vizsgálat eredményeit összegző táblázatból látható, hogy első körben 4 benchmark módszert választottam ki: a BPMN Miner, a Decomposed Process

Miner, az Evolutionary Tree Miner és az Inductive Miner eljárásokat, mert ezek adják vissza legpontosabban a folyamat vezérlési menetét (control-flow). Mivel azonban az OR-joint is tartalmazó BPMN modellek és Petri-hálók esetén a pontosságot és az illeszkedés mértékét nem tudjuk mérni (nem találtam hozzá módszert), ezért a folyamatleíró fát visszaadót eljárásokat fogjuk referenciaként használni: az Evolutionary Tree Minert és az Inductive Minert.

ESEMÉNYSOROK GYAKORI MINTÁINAK FELTÁRÁSA GRÁFALAPÚ MÓDSZERREL

DR. RADELECZKI SÁNDOR

A projekt keretében az én feladatom általában a matematikai modellezés, ezen belül, a gráf/hálózatalapú folyamatfeltárás területén:

- *diszkrét matematikai módszerek elemzése és rendszerezése, az eseménysorok gyakori mintáinak feltárása;*
- *módszerek kidolgozása az eseménysorok gyakori mintáinak a feltárására;*
- *gráfokon alapuló modellek leírásának dokumentálása és ezeknek a módszereknek az elemzése;*
- *algoritmusok megfogalmazása, és az elvégzett elemzések dokumentálása.*

1. A kutatás célja és lépései

Első lépésként elvégeztem a témakörben fellelhető jelentősebb cikkek elemzését. Megvizsgáltam és rendszereztem az ezekben a publikációkban szereplő fogalmakat és eszközöket/módszereket. A szakirodalomban több, (egymással is kapcsolódó) módszerrel, megközelítéssel találkozhatunk, amelyeket egymással összehasonlítottam és kiemeltem közülük az alábbi csoportot:

Gráfalapú módszerek a leggyakoribb minták felismerésére – ezeknek több változata is ismert, az egyik legkorszerűbb módszer közülük a maximális méretű gyakori mintákat tárja fel egy véges parciális automata állapotgráfjának a megvalósítása által (Maximal Pattern Mining). Itt kiindulópontom a Liesaputra, V., Yongchareon, S. and Chaisiri, S.: (2016, Sept.). Efficient process model discovery using maximal pattern mining. In *International Conference on Business Process Management*, pp. 441–456), Springer, Cham. című publikáció volt.

Második lépésként rámutattam ennek az ún. MPM- (maximal pattern mining) módszernek az előnyeire a többi gráfokon alapuló módszer viszonylatában. Elemeztem az MPM-módszer szubrutinjait és implementálásra alkalmas verzióit. Ennek alapján vázoltam, hogy egy adott tevékenységfolyamnál hogyan határozható meg a fő tevékenységsor, a lehetséges elágazások, a gyakorisági paraméterek és az esetleges kivételek, illetve azt, hogyan becsülhető meg a módszer „pontossága". Megvizsgáltam, hogy a megadott algoritmus hogyan finomítható tovább és kiegészítettem az utóbbi néhány év erre vonatkozó irodalmával.

Részletesen megvizsgáltam a módszer lehetséges értékelési kritériumait (beleértve a pontosságot) és ezeknek a mérőszámait. Megvizsgáltam, hogy hogyan finomítható tovább úgy (MPM) algoritmus (lásd [2]), hogy a hurkok hosszát is nyilvántartsa és egyszerűsítse az eredményként kapott tranzíciós gráfot.

Elvégeztem az MPM-algoritmus néhány (az irodalomban nem kellően részletezett) része implementációjának az előkészítését, az eljárások leírásával és pontosításával. Megadtam az eljárás egyes lépéseihez tartozó szubrutinok pszeudokódjait és a keretalgoritmus pszeudokódját. Módszert fogalmaztam meg optimális megoldások megkeresésére. Ezt követően implementáltam a teljes algoritmust.

Kipróbáltam az MPM-algoritmus szubrutinjainak és magának a teljes algoritmusnak is a működését, lefuttatva egyszerűsítek példák egy kisebb halmazán. Az egyes szubrutinokat külön-külön elemeztem.

2. Kutatási eredmények összesítése

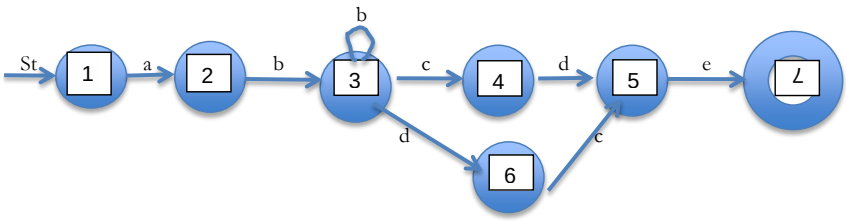
2.1. A bemutatott szakirodalom elemzése, értékelése, alapfogalmak és módszerek kiemelése

Egy vállalat belső ügyfélszolgálati rendszerének egy cselekményét *nyomnak (trace)* vagy *akciónak (action)* nevezzük. Egy t *nyom (trace) események (events)* egy véges és koherens $z_0, z_1, \ldots, z_m$ sorozata, amit a $t = (z_0, z_1, \ldots, z_m)$ formában jelölünk. Az egyes z_i eseményeket vektorként ábrázoljuk, ami kötelező komponensként kell tartalmazza az esemény *típusát (type)*, valamint *időbélyegét (timestap),* más egyéb adatok mellett. A t nyomon belül az egyes eseményeket időbélyegük szerint rendezzük sorba. Egy t nyomban csak az őt alkotó események típusait ábrázoljuk, pl. t = (a, b, b, c, e, d). Így egy véges ábécé feletti szavakat kapunk (ahol az „ábécé" az eseménytípusok halmaza), egy T = {(a, b, c, b, b, c, d, e), (a, b, b, c, b, c, d, e), (a, b, b, c, e, d)} eseménynaplót pedig egy (véges ábécé feletti) formális nyelvnek (vagy egy ilyen nyelv egy részletének) tekinthetünk. Egy t nyom *hossza*, amit |t|-vel jelölünk, nem haladhat meg egy előre megadott korlátot.

Feladatunk most úgy fogalmazható meg, hogy egy lehetséges eseménynaplóban egyrészt ellenőrizni akarjuk a cselekmények nyomait, megkülönböztetve őket a *zajtól* (ami itt egy nyom eseményeinek hibás vagy hiányos rögzítéséből adódhat), illetve olyan nyomokat szeretnénk automatikus módon generálni, amik valódi ügyfélszolgálati eseményeknek felelnek meg.

Az irodalomból itt a feladat megoldására két lehetséges módszert (megközelítést) emeltem ki:

1) Az első megközelítés esetén (α++- algoritmus, MPM [maximal pattern mining] algoritmus, OSTIA) a nyomok hasonló szerkezetű csoportjaihoz egy tranzíciós gráfot rendelünk – ezt egy címkézett és irányított gráfnak tekinthetjük és *tranzakciós mintának (transaction pattern)* nevezzük. A MPM-algoritmus esetén először maximális mintákat választunk ki, amik részgráfként tartalmazzák a többi mintát és ehhez rendeljük egy véges determinisztikus parciális automata tranzíciós gráfját



2.1. ábra. *$T = \{(a, b, c, d, e), (a, b, d, c, e), (a, b, b, c, d, e)\}$ mintához tartozó véges automata*

2) A második megközelítés esetén megerősítő tanuláson alapuló szövegfelismerést és szöveggenerálást alkalmaznak, ami egy összetett, rekurens neuronhálózat segítségével valósul meg. A tanulás során egy előre kiválasztott tanítóhalmaz alapján először a leggyakoribb nyomokkal megegyező szerkezetű nyomokat állítjuk elő, majd fokozatosan a ritkább előfordulású nyomokat is „reprodukáljuk".

A gyakorlatban az ügyfélszolgálati rendszer a beérkező valós *kérésekre (request)* kell (megoldásokat) válaszokat adjon. Egy ilyen r kérés (igénylés) maga is egy vektor formájában adható meg, ami tartalmazza a kérés típusát, az igénylő azonosítóját, a beérkezés (iktatás) idejét, valamint a kéréssel kapcsolatos szignifikáns adatokat. Az ügyfélszolgálati rendszer erre egy eseménysorral válaszol, ami az eseménynaplóban egy nyomként szerepel. Itt a legfontosabb információ a nyom „mintája", ami az első esetben egy véges automatának, az automatikus szövegfelismerés esetén pedig „rokon értelmű" szavak egy halmazának felel meg. Az ERPA öntanuló rendszer tanítóhalmazában tehát (r_i, t_i) kérés-nyom pároknak kell szerepelniük. Hogy ez elkészüljön, ezt meg kell előzze a beérkező kérések megfelelő (dimenzió-csökkentett) formában való rögzítése és különböző szempontok szerinti automatikus osztályozása.

Az első megközelítés esetén, a tanítóhalmaz alapján az (r_i, t_i) párokhoz rendelt véges automaták „finomhangolása" végezhető el, vagyis a tanítóhalmazban szereplő példák alapján, a t_i mintához tartozó véges automatának a tranzíciós diagramjában a programnak meg kell találnia a kérést megválaszoló legjellemzőbb útvonalat. A második megközelítés esetén a tanítóhalmazban szereplő (r_i, t_i) párok alapján az automatikus szöveggeneráló rendszernek elő kell állítania a leginkább jellemző „kérdés-felelet" típusú szövegrészleteket. Ezeknek a tökéletesítése megerősítő tanulás által érhető el. A továbbiakban az MPM-módszerre fókuszáltam.

2.2. Az MPM-módszer bemutatása és elemzése.

1. Az eseménynapló alapján a fő tevékenységsorokat úgy kaphatjuk meg, hogy a kibányászott maximális gyakoriságú sémákból a hurkokat töröljük. (Az ismétlődő szekvenciák nem változtatnak a folyamat természetén.) A gyakoriság figyelembevétele azért fontos, hogy a zajt és a kivételes eljárásokat kiszűrhessük. A nyomokban szereplő hurkok kiküszöbölése két lépésben történik:

a) az ismétlődő típusokat (karaktereket) egyetlen karakterrel helyettesítjük. A fenti példa esetén az (a, b, c, b, c, d, e), (a, b, c, b, c, d, e), (a, b, c, d, e) kifejezéseket kapjuk.

b) Ezután az ismétlődő karaktercsoportokat egyetlen csoporttal helyettesítjük, itt például: (a, (b, c, d, e), (a, (b, c, d, e), (a, b, c, d, e). Az utolsó kifejezés nem változik, az első kettő pedig ugyanaz lévén, elég egyszer feltüntetni, így kapjuk, hogy: (a, (b, c, d, e), (a, b, c, d, e). A * jeleket (az exponenseket) elhagyjuk a kapott kifejezésekből és megadjuk a hurkok helyét (végét) és a bennük lévő betűket. Így megkapjuk a T csoporthoz tartozó ún. *sémát* (pattern) – ez mindhárom esetben: $p = \langle a, b\langle b\rangle, c\langle bc\rangle, d, e\rangle$. A **p** séma tartója (support-ja) nem más, mint a **p**-hez tartozó nyomok száma: ez most 3.

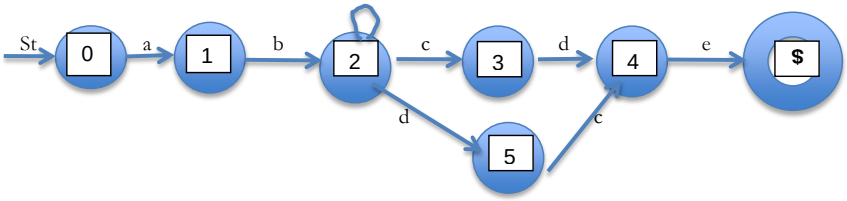
2. A gyakori műveletsorok (eseménysorok) feltárása az MPM-eljárásban két küszöbérték (trashold érték) megadásával lehetséges, úgy ahogy azt a korábbi összefoglaló jelentésben bemutattuk, felhasználva a nyomok ún. „függőleges bitmap reprezentációját". Ennek a reprezentációnak a segítségével a sémák listáján való egyszeri végighaladással leolvasható az egyes karakterek előfordulási számai és azokat a karaktereket (típusokat), amelyeknél ezek relatív előfordulása nem haladja meg a trash szám értékét, töröljük az eseménynaplóból azokkal a sémákkal és nyomokkal együtt, amelyekben előfordulnak. Ahhoz, hogy a kivételes eseménysorokat a zajtól megkülönböztessük, szükségünk van mind a bejövő kérések ismeretére, mind egy nagy elemszámú validációs halmazra. Ezután a gyakori sémák kiválasztása két különböző módon is történhet. Az egyik kézenfekvő módszer az egyes sémákhoz vezető nyomok tárolása. Végighaladva a sémák listáján, minden egyes séma esetén összeszámoljuk azokat a nyomokat, amiket az adott sémák „lefednek". Ha egy séma esetén ezeknek a relatív száma meghaladja a trash értéket, akkor a sémát megtartjuk, ellenkező esetben a sémát és a hozzátartozó nyomokat töröljük. Egy másik módszer egy az idézett cikkben megadott *Expand subroutin* ami a gyakori karakterekhez iteratív módon mindig egy-egy karaktert hozzácsatolva generálja

a gyakori sémákat. Az ebben a 2. lépésben szereplő két trash értéket egy egyszerű tanulási folyamat során tudjuk optimálisan beállítani.

3. A módszer két különböző típusú lehetséges elágazást tud felismerni. Ha közös kezdő és végső részszekvenciával rendelkező eseménysorok, például $p_1 = us_1v$, $p_2 = us_2v$, $p_3 = us_3v$, $p_4 = us_4v$ csak egy-egy részszekvenciában különböznek, azt egy formális összegként az $u(s_1 + s_2 + s_3 + s_4)v$ reguláris kifejezéssel írhatjuk le, ami annyit jelent, mint ha az $s_1, \ldots, s_4$ részszekvenciákat a logikai *vagy* (diszjunkció) kötné össze és ezt grafikusan egy (négyes) formális elágazással szemléltethetjük.

(1) Ez akkor jelent valóban opcionális helyzetet, ha a $p_1, \ldots, p_4$ sémák ugyanarra a kérésre adott válaszok (nyomok) összeségét jelentik. Ekkor a $p = (u, XOR(s_1, s_2, s_3, s_4), v)$ sémát kapjuk, ami valódi elágazást jelent.

(2) Párhuzamos helyzet. Egy másik sajátos eset az, ha $s_1, \ldots, s_4$ olyan szekvenciák, amelyek egy karaktercsoport permutációi, méghozzá úgy, hogy bármely két karakter fordított sorrendben is megjelenik a példákban. Ilyen például az $s_1 = (a, b, c)$, $s_2 = (a, c, b)$, $s_3 = (b, a, c)$, $s_4 = (c, b, a)$ eset. Ekkor a mintákból arra következtetünk, hogy az a, b, c események sorrendje irreleváns és azok párhuzamosan is végrehajthatók. Ekkor az $(u, \{a, b, c\}, v)$ sémát kapjuk eredményül, ami rajzban szintén elágazással szemléltethető. Például az (a, b, c, d, e) és az (a, b, d, c, e) sémák esetén a c és d események tetszés szerinti sorrendben elvégezhetők. Ezért mind a kettő valójában egyetlen folyamathoz tartozik, aminek a sémáját így jelöljük: (a, b, {c, d}, e). A {c, d} halmazjelölés arra utal, hogy a c és d események sorrendje mellékes. Más jelölés: (a, b, ∧c, d), e). A sémának megfelelő véges parciális automata pedig az alábbi.



2.2. ábra. *Az (a, b, b, c, d, e) és (a, b, d, c, e) nyomokhoz tartozó véges automata*

Az alapcikkben a párhuzamosságok feltárását egy Solve Concurrency nevű algoritmus végzi. Az algoritmus csoportosítja azokat a sémákat, amelyek egy közös

prefixszel, sufixszel, vagy mindkettővel rendelkeznek és kiemeli a nem közös részt, amennyiben azon belül nincs ismétlődő csoport

4. Negyedik lépésként az algoritmus az ún. maximális sémákat keresi ki és listázza. Az mondjuk, hogy a p_1 sémát *lefedi* a p_2 séma, ha minden, a p_1 sémával előállítható nyom, a p_2 sémával is megkapható. Így például az (a, b, c, d, e) és az (a, b, d, c, e) sémák lefedhetők az (a, b, {c, d}, e) sémával, de az (a, b, c) séma nem (noha a hozzátartozó gráf részgráfja lenne az (a, b, {c, d}, e) sémához tartozó tranzíciós gráfnak). Egy T csoporton belül egy p séma *maximális*, ha nincs olyan tőle különböző q séma T-ben, ami p-t lefedi. (Tehát nincs olyan q séma, ami mindazokat a nyomokat előállítaná, amit p – és esetleg még azon felül mást is.) A Resolve szubrutin ebben a lépésben kettesével összehasonlítja a tanítóhalmazból kinyert, egy IdList nevű listán szereplő p_i és p_j sémákat egy kettős iteráció során. Ha a p_j séma lefedi a p_i-t akkor p_i helyére p_j-t ír és p_i-t törli. Ezután folytatja p_j összehasonlítását az eddig nem vizsgált sémákkal. Az eljárás végén az algoritmus listázza a megmaradt sémákat, amelyek már mind maximálisak.

5. Az utolsó lépésben az algoritmus megrajzolja az eseménynaplóban adott munkafolyamathoz tartozó *tranzíciós gráfot (folyamatgráfot)*, amit a (kibányászott) maximális sémákhoz tartozó parciális véges automaták tranzíciós gráfjainak az uniójaként kap meg. A kapott véges automaták kezdőállapotait egyetlen kezdő (Start) állapotban, végállapotait egyetlen, $-ral jelölt végállapotban egyesíti.

2.2.1. A zaj *kiküszöbölése tanulással és az* „eredményesség" mérése

Azokat az eseménytípusokat, illetve azokat a sémákat, amelyek gyakorisága a megfelelő trash értéket nem haladja meg, *zajnak* tekintjük és kiküszöböljük. A trash paraméterek „optimális" beállítása úgy történik, hogy a megadott nyomok M összeségét két diszjunkt (nem üres) halmazra, egy ún. T *tanítóhalmazra* és egy V *validációs* vagy *teszthalmazra* osztjuk. Az algoritmus tesztelése során előállított nyomok halmazát R-rel jelöljük. Az ún. eredményesség mérésére az algoritmus több mérőszámot is használ. Itt a legfontosabbat ismertetjük:

Az egyik az ún. *Fitness* vagy *Recall* mérőszám (*Érvényesség, Visszaidézés*) – ez nem más, mint a helyesen előállított nyomok számának (tehát az $R \cap V$ halmaz elemszámának) és az eseménynaplóban szereplő nyomok M számának az aránya, vagyis

$$\Re := \frac{|R \cap V|}{|M|}$$

A *Precision („Pontosság")* mérőszám a helyesen előállított nyomok $(R \cap Vhalmaz)$ elemszámának és az összes előállított nyom (R) elemszámának az arányát jelenti:

$$P := \frac{|R \cap V|}{|R|}$$

Az ún. *F-mérték (F-measure)* egy, a két előbbi mutatót összesítő globális mutató, ami tulajdonképpen *Re* és *P* harmonikus középarányosa. Ezért

$$F_m := \frac{2 * P * \Re}{P + \Re}$$

Egy ritkábban használt mérőszám az ún. Eltérés vagy Különbség (Difference), ami az *R* és *V* szimmetrikus különbségének és (R) elemszámának a hányadosa.

$$D := \frac{|R \Delta V|}{|R|}$$

Természetesen ezek a mérőszámok nem függetlenek egymástól. Egy másik mérőszám az a **t** *idő*, ami alatt az (MPM) algoritmus felállít egy folyamatmodellt (valójában az egyszerűsített tranzíciós gráfot). A trash paraméterek „optimális" beállítása úgy történik, hogy a megadott nyomok M összeségét két diszjunkt (nem üres) halmazra, egy ún. T *tanítóhalmazra* és egy V *validációs* vagy *teszthalmazra* osztjuk. Az MPM-algoritmus a tanítóhalmaz alapján előállítja a maximális sémákat. Aztán megvizsgálja, hogy az így kapott sémák által előállított nyomok R halmaza milyen mértékben egyezik meg a V teszthalmazzal. Ha a $V - R$ halmazkülönbség relatív elemszáma nagy, akkor a trash értékeket csökkentjük. Ha az $R - V$ halmazkülönbség relatív elemszáma nagy, akkor a trash értékeket növeljük. Ezt mindaddig folytatjuk ameddig az R és V halmazok a lehető legkisebb számú elemben különböznek. Valójában az MPM-eljárás a trashold értékek beállítására és a fenti mérőszámok kiszámítására, az ún. *k-szoros validációt* használja. Ez azt jelenti, hogy a rendelkezésünkre álló nyomhalmazt k drb. diszjunkt (és nagyjából azonos elemszámú) részhalmazra bontjuk (Itt $k \geq 3$). Először az 1. számú részhalmazt jelöljük ki tanítóhalmaznak, a többiek uniója pedig a teszthalmaz lesz. Ezután a 2. részhalmaz lesz a tanítóhalmaz és a többiek uniója a teszthalmaz, …, ezt

mindaddig folytatjuk ameddig mind a k részhalmaz sorra kerül, mint tanítóhalmaz. Minden egyes lépésnél meghatározzuk az ott adott felosztásra vonatkozó mérőszámokat. Ezek számtani középarányosaiként kapjuk meg az egész eljárásra vonatkozó mérőszámokat (jelen esetben a Fitness és a Precision mérőszámokat).

2.2.2. A kivételek meghatározása

A kivételek olyan kis gyakorisággal előforduló minták, amik megkülönböztethetők a zajtól. Jó eséllyel a beérkezett kérések (requests) ismeretében különíthetőek el, amelyek jól megkülönböztethetőek kell legyenek a többi kéréstől. A tanulási folyamat során ellenőrizzük, hogy a validációs halmazban a kitüntetett requesteknek megfelelő nyomokat megkapjuk-e, ha nem, akkor a tanulási folyamatot tovább folytatjuk.

2.3. Az elemzések eredményeinek bemutatása

2.3.1. Az ismétlődések reprodukálása

Amint azt már korábban mondtuk, az MPM-eljárás első lépéseként kiküszöböljük a nyomokban szereplő hurkokat. A [2] cikkben az ismétlődő típusok vagy típuscsoportok jelölésére zárójeleket javasolnak. Például az $a, b\langle b\rangle, c, d$ jelölés azt jelenti, hogy a b típusú esemény többször is ismétlődhet, de legalább egyszer elő kell forduljon. Ha $a, \langle b\rangle, c, d$ -t írunk, akkor b hiányozhat is. A $p = \langle a, b\langle b\rangle, c\langle bc\rangle, d, e\rangle$ minta esetén a bc csoport tetszőleges sokszor fordulhat elő. A formális nyelvek jelölését használva az ismétlődés pontos számát is jelölni és a memóriában tárolni is tudjuk, pl. (a, b^2, c, d, e). Általában ez az ismétlődési szám egy ügyviteli folyamatban nem haladhat meg egy előre megadott felsőkorlát-számot, ami legtöbbször egy kis szám. Egy ismétlődő eseménytípus (pl. $\langle b\rangle$) esetén ez viszonylag könnyen megoldható valamilyen „klaszterezési" eljárással a következőképpen: Minden (a treshold értéket meghaladó gyakoriságú) t_i nyom esetén tároljuk az ismétlődő eseményhez (itt pl. a b előfordulásaihoz) tartozó számszerű paramétereket egy $f_i = \langle p_1, .., p_{k(i)}\rangle$ vektorban. Ez pontosabban azt jelenti, hogy minden olyan (a, b, c, d, e), (a, b^2, c, d, e), (a, b^3, c, d, e) nyom esetén (aminek a gyakorisága nagyobb a treshold értéknél) annyi $\langle p_1, .., p_{k(i)}\rangle$ alakú vektort tárolunk amennyiszer az adott nyom (pl. (a, b, c, d, e)) előfordul, majd valamilyen ismert osztályozási eljárással, pl. K-mean, Support vector machine, az 1, 2, 3,… előfordulásoknak megfelelő diszjunkt csoportokra osztjuk őket. Így egy újabb beolvasott nyom esetén már megvizsgálhatjuk, hogy a b-hez tartozó paramétervektor besorolható-e valamelyik osztályba – a válasz akkor lesz igen, ha az új paramétervektor távolsága az adott osztály átlagától nem haladja meg az

osztályon belül az átlaghoz (ami valójában a klaszter középpontja) viszonyított távolságok közül a legnagyobbikat. Egy ismétlődő csoport (pl. $\langle bcd \rangle$) esetén, annyival bonyolultabb a helyzet, hogy itt mind a b, mind a c, és mind a d eseményekhez tartozó paraméter-vektorokat ismernünk kellene és belefoglalni őket egyetlen paraméter-vektorba; természetesen nem minden paraméter fogja az ismétlődést befolyásolni, ezért a paraméterek számának túlzott növekedését a nem releváns paraméterek kiküszöbölésével csökkenthetjük. (Az algoritmus futási ideje ugyanis a vizsgált vektorok hosszával hatványozottan nő!) Erre, valamilyen egyszerű statisztikai eljárás lehetne alkalmas, például a bcd csoport előfordulásainál megvizsgálnák az egyes paramétereknek a várható értékét és szórását. Ahol a teljes ismétlődéssorozatra kapott szórás „nagy" lenne – vagyis a várható értékhez hasonló nagyságú lenne, azokat a paramétereket nem vennénk figyelembe és nem írnánk be a paramétervektorba. Ez az eljárás természetesen továbbfinomítható, ha már az „alapprogram" elkészül és működik.

Egy másik, itt most csak megemlített lehetőség, az a helyes paraméter-vektorok neurális hálókon alapuló „megtanulása" lenne, vagyis annak a megerősített tanulással való behatárolása, hogy egy új paramétervektor az esemény, vagy az esemény-csoport hány ismétlődéséhez kapcsolható hozzá. Itt eredményesen alkalmazhatnánk a belső jelentésekben bemutatott ún. bitmapes reprezentációját az egyes nyomoknak.

2.3.2. Optimális útvonalak keresése

Ahhoz, hogy egy tranzíciós gráfon belül ún. optimális útvonalakat jelöljünk ki, tehát döntsünk az esetleges ismétlődések számáról és az opcionális részszekvenciák kiválasztásáról, szükséges lenne a kérések és a kibányászott minták között kapcsolatot ismernünk. Ez elérhető akkor, ha a tanítóhalmazban kérés-nyom párok szerepelnek – ahol a nyom a kérésre adott válaszhoz tartozik. Ekkor egy adott kéréshez egy célfüggvényt rendelve, rendezhetjük a hozzá tartozó (válaszként kapott) nyomokat – és kiválaszthatnánk az optimális megoldást jelentőt közülük.

2.3.3. A folyamatgráf megrajzolása és egyszerűsítése

Az eljárás végén kapott tranzíciós gráfot (az opcionális és párhuzamos szekvenciák meghatározásával és a kivételes minták elkülönítésével) egyszerűsíthetjük. Az eljárás végén a tranzíciós gráfot (folyamatgráfot) úgy rajzoljuk meg lépésről lépésre, hogy a kinyert maximális mintákhoz (sémákhoz) parciális véges determinisztikus automaták tranzíciós diagramjait rendeljük, majd az eljárás végén ezeket egyetlen gráfban vonjuk össze, azaz egyesítjük. Ez a gyakorlatban a Python programhoz tartozó könyvtárak közül a „pytranzition" könyvtárban található rajzprogram

segítségével történik, ami alkalmas automaták tranzíciós diagramjainak és Petri-hálók diagramjainak a megrajzolására is: Pytransitions függvénykönyvtár és útmutató: https://github.com/pytransitions/transitions/blob/master/README.md

2.3.4. Eredményesség mérése

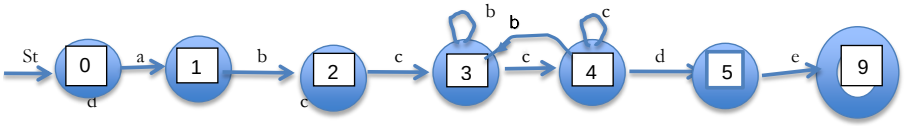
Az algoritmus „eredményességének" mérésére több (egymástól nem független) mutató is alkalmazható. Itt négy ilyen mérőszámnak (Érvényesség, Pontosság, F-mérték, Különbség) is megadtuk a pontos matematikai alakját. Itt is elmondható, hogy az eredmények helyes kiértékelése a valódi ügyviteli folyamatot megadó tanítóhalmazok alapján végezhető el hatékonyan.

3. Az MPM-eljárás technológiája: a szubrutinok leírása és pszeudokódjaik megadása

3.1. Az ismétlődések (hurkok) feltárása

Ez az eljárás a tevékenységnaplóban megadott nyomokban meghatározza az ismétlődéseket. Bemenete a vizsgálandó nyom, kimenete pedig a keletkező minta, **(pattern)**, amelyben az ismétlődések (hurkok) jelölve vannak. A ciklusok feltárását a SOLVE_LOOP névvel hivatkozott eljárás végzi.

Például az $\langle a, b\langle b\rangle, c\langle bc\rangle, d, e\rangle$ minta esetén a *bc* csoport többször is előfordulhat, vagy egyszer sem. Egy nyomot események egy sorozatának tekintünk, amelyeket 0-val kezdve indexelünk. A nyom egy elemére az indexével hivatkozunk. Például a $t = \langle a, b, c, b, b, b, c, b, c, b, c, c, d, e\rangle$ nyom esetén $t[4] = b$. A nyom egy szakaszát elejének és végének az indexét megadva jelöljük és közéjük két pontot teszünk, pl. $t[1..4] = \langle b, c, b, b\rangle$. Először az ismétlődő karaktereket szűrjük ki, $\langle a, b, c, \langle b\rangle, c, b, c, b, \langle c\rangle, d, e\rangle$**,** majd az ismétlődő csoportokat is megjelölve az ismétlődés helyét $\langle a, b, c, \langle\langle b\rangle, \langle c\rangle\rangle, d, e\rangle$**.** Most az ismétlődéseket, vagyis a $\langle \quad \rangle$ zárójeleket elhagyva, megkapjuk a mintát $p = \langle a, b, c, b, c, d, e\rangle$ ami a nyomhoz tartozik. Ahhoz, hogy ennek alapján elkészítsük a nyomnak megfelelő tranzíciós gráfot – azaz egy véges automatát, az ismétlődések pontos helyét is tárolnunk kell. Itt ezek helyét az eredményül kapott *p* mintára vonatkozó indexeléssel adjuk meg. Ezt egy intervallumhalmazzal adjuk meg, ahol egy intervallum kezdő- és végpontja azt mutatja, hogy hányadik elemtől hányadik elemig ismétlődnek az elemek a mintában (sémában). Az előző példa esetén ez az intervallumhalmaz a következő: $\{(3,3), (4,4), (3,4)\}$. Az ennek megfelelő automata az alábbi parciális véges determinisztikus automata lenne.



Az ismétlődések kijelölésekor arra kell ügyelnünk, hogy elkerüljük azt ún. *ütköző* intervallumokat. Az ütközésmentes intervallumokat úgy definiáljuk, hogy egy intervallum vagy teljesen benne van egy másikban, vagy egyáltalán nincs közös elemük. Például, az $\{(1,4),(2,4),(5,7)\}$ halmaz ütközésmentes, viszont az $\{(1,4),(3,5),(5,9)\}$ intervallumhalmazban két ütközés is található.

Az algoritmus először az 1-es hosszúságú ismétlődéseket keresi, tehát amikor egy karakter ismétlődik, majd ennek eredményén a 2-es hosszúságúakat, és így tovább, egészen addig, amíg meg nem haladja a hossza a minta méretét. A fenti példában először az $\langle a,b,c,\langle b\rangle,c,b,c,b,\langle c\rangle,d,e\rangle$ eredményt kapjuk, majd a 2-es hosszúságú ismétlődésekre az $\langle a,b,c,\langle\langle b\rangle,\langle c\rangle\rangle,d,e\rangle$ eredményt. Az eljárás egy ún. *„ablakot”* mozgat balról- jobbra az adott nyomon (azaz előző iterációban kapott mintán), és azt vizsgálja, hogy az ablak tartalma ismétlődik-e közvetlenül utána. Ha igen, az ismétlődő részt csak egyszer veszi fel az eredménymintába, illetve felveszi az ablakot az ismétlődések halmazába. Itt az $a,b,cprefixbőlabc$ csoport azért nem került be az ismétlődő csoportba, mert az eredeti nyomban az ismétlődés nem közvetlenül a c karakter után kezdődött, hanem eggyel később (a 3-as index után).

SOLVE_LOOP(trace)

```
pattern = []
loops = ∅
FOR len in {1, …, trace.length}
        newPattern = []
        newLoops = ∅
        FOR i in {0, …, trace.length - 1}
                window = trace[i .. i+len-1]
                currentLoop = NULL
                firstIter = true;
                ni = i + length
                WHILE true
                        next = trace[ni .. ni+len-1]
                        checkInterval = (ni, ni+len-1)
                        tloops = ∅
                        FOR loop in loops
                                IF loop ⊆ checkInterval AND currentLoop ≠ NULL
                                        tloop = translate loop from
```

```
                                                    checkInterval into currentLoop
                                                IF tloop does not collide with any "subloops" of
                                                currentLoop
                                                        tloops = tloops ∪ {tloop}
                                    IF window = next
                                            IF firstIter
                                                    newPattern.add(window)
                                                    newLoop = (newPattern.length – len,
newPattern.len - 1)

                                                    newLoops = newLoops ∪ {newLoop} ∪
                                                    {tloops}
                                            currentLoop = newLoop
                                            nextIndex = nextIndex + len
                                            firstIter = false
                                    ELSE
                                            IF firstIter
                                                    newPattern.add(current[0])
                                                    i = i + 1
                                            ELSE
                                                    i = nextIndex
                                            EXIT WHILE
        pattern = newPattern
        loops = newLoops
    RETURN pattern, loops
```

3.2. A zaj(ok) kiküszöbölése

Az eljárás az eseménynaplóban található nyomok közül eltávolítja a küszöbértéket meg nem haladó relatív gyakoriságú eseményeket, illetve ezután a ritka nyomokat (tehát a zajnak tekintett karaktereket és nyomokat). A ritka események eltávolítása után végighaladunk a nyomok listáján és minden egyes mintát minden nyommal összehasonlítunk, megvizsgálva, hogy illeszkedik-e a nyom az adott mintára. Minden lépésnél, tehát minden minta esetén tároljuk azokat a nyomokat, amik erre a mintára illeszkednek (tehát előállíthatók a mintához tartozó véges automatával) és a számukat is. Megvizsgáljuk, hogy melyek azok a minták, ahol ennek a számnak és az összes nyom számának az aránya meghalad egy előre megadott α küszöbszámot, azaz trash értéket (például $\alpha = 0{,}01$ - et). A küszöbszám értékének a beállítása valójában a tanulási folyamat során a validációs halmaz nyomait felhasználva történik meg. A trash értéket nem elérő relatív gyakoriságú mintákat zajként kezeljük és töröljük a minták listájáról.

```
1.  REMOVE_NOISE(eventlog, thresh)
2.          occurences = []
3.          result = copy of eventlog
4.          FOR each trace in eventlog
5.                  FOR each event in trace
6.                          occurences[event] = occurences[event] ∪ {trace}
7.          FOR each event in occurences.keys
8.                  IF occurences[event].count / eventlog.count < thresh
9.                          result = result \ { trace in eventlog | trace contains event }
10.         RETURN result
```

3.3. Vizsgálatok

3.3.1. Párhuzamosságok feltárása (a minták között)

Az eljárás bemenete két minta (p_1 és p_2), amelyek között az algoritmus azonosítja a párhuzamosságot. Amennyiben nincs köztük lehetőség párhuzamosság jelölésére, akkor az algoritmus visszatér mindkét mintával. Ha a két minta között van párhuzamosság, akkor egyetlen egyesített mintával, mint eredménnyel tér vissza. Az eljárás a következő fő lépéseket tartalmazza:

- Megállapítjuk, hogy a minták milyen indexig egyeznek meg az elejéről nézve.
- Megállapítjuk, hogy a minták milyen indexig egyeznek meg a végéről nézve.
- Ellenőrizzük, hogy a két index között ugyanazok az elemek szerepelnek-e, vagy az egyik elemhalmaz része-e a másiknak.
- H nincs a két index közt elem, vagy az ott szereplő elemekre a fenti feltétel nem teljesül, akkor nem beszélhetünk párhuzamosságról.
- Ha ugyanazok az elemek szerepelnek a két index között (más sorrendben), vagy az egyik elemhalmaz része-e másiknak, akkor megvizsgájuk, hogy bármelyik két karakter, illetve a két csoport sorrendje felcserélhető-e. Ha nem, mert nincs minden inverzióra példa, akkor nincs párhuzamosság sem.
- Ha a válasz igen, akkor megállapítjuk a párhuzamosság tényét és egy kombinált mintát hozunk létre, amellyel visszatérünk az eljárásba.

A mintában a párhuzamos elemeket egy halmazzal jelöljük – ezzel kifejezve, hogy a sorrendjük nem számít. A már említett $p = \langle a, b, c, d, e \rangle$ és $q = \langle a, b, d, c, e \rangle$ minták esetén van, a kapott eredmény pedig $\langle a, b, \{c, d\}, e \rangle$ lenne.

```
1.  SOLVE_CONCURRENCY(p, q)
2.      IF p.length ≠ q.length
3.          RETURN p, q
4.      start = -1
5.      end = p.length
6.      FOR i in {0, 1, … p.length - 1}
7.          IF p[i] ≠ q[i]
8.              start = i – 1
9.              EXIT FOR
10.     FOR i in {p.length-1, p.length-2, …, 0}
11.         IF p[i] ≠ q[i]
12.             end = i + 1
13.             EXIT FOR
14.     IF start > end
15.         RETURN p, q
16.     FOR i in {start+1 .. end-1}
17.         cp = 0
18.         cq = 0
19.         FOR j in {start+1 .. end-1}
20.             IF p[i] = p[j]
21.                 cp = cp + 1
22.             IF p[i] = q[j]
23.                 cq = cq + 1
24.         IF cp ≠ cq
25.             RETURN p, q
26.     RETURN ⟨p[0], …, p[start-1], {p[start], …, p[end]}, p[end+1], …, p[p.length-1]⟩
27.
```

3.3.2. Opcionális vizsgálata (a minták között)

Az eljárás bemenete az előzőleg kapott minták listája. Az eljárás először minden minta-párra megállapítja, hogy hány egyezés van az elejükön (prefixükön) és a végükön (szufixükön). Ahol az egyezések száma a legnagyobb, azt a két mintát egyesíti úgy, hogy az egyező részeket az elején és a végén meghagyja, a köztes részt pedig összekapcsolja XOR-ral. A két mintát eltávolítja a listából, és az egyesített változatot hozzáadja, majd rekurzívan az algoritmus újra indul a kapott mintalistára. Ha a minták listája egyelemű, nincs mit egyesíteni, az algoritmus megáll. Ha nincs egyezés a listában szereplő minták között sehol, az összeset összekapcsolja egy XOR-ral (kijelölve egyetlen kezdő- és végállapotot). Így az algoritmus eredménye mindenképpen egyetlen minta lesz.

```
1.  SOLVE_OPTIONALITY(pList)
2.      IF pList.length = 1
3.          RETURN pList
4.      starts = [][]
5.      ends = [][]
6.      FOR i in {0, 1, … p.length - 1}
7.          FOR j in {i + 1, I + 2, …, p.length - 2}
8.              maxindex = min(pList[i].length, pList[j].length)
9.              FOR k in {0, …, maxindex - 1}
10.                 IF pList[i][k] ≠ pList[j][k]
11.                     starts[i][j] = k
12.                     EXIT FOR
13.             FOR k in {maxindex - 1, …, 0}
14.                 IF pList[i][k] ≠ pList[j][ k]
15.                     ends[i][j] = maxindex - k
16.                     EXIT FOR
17.     u, v = argmax{starts[u][v] + ends[u][v] | u,v in {0, …, pList.length}, u < v}
18.     IF starts[u][v] + ends[u][v] = 0
19.         RETURN XOR(pList)
20.     ELSE
21.         result = take first starts[u][v] elements of pList[u].
22.         x = pList[u]
23.         Remove first starts[u][v] elements from x.
24.         Remove last ends[u][v] elements from x.
25.         y = pList[v]
26.         Remove first starts[u][v] elements from x.
27.         Remove last ends[u][v] elements from x.
28.         Append XOR(x, y) to result.
29.         Append last ends[u][v] elements of pList[u] to result.
30.         Remove pList[u] and pList[v] from pList.
31.         Add result to pList.
32.         RETURN SOLVE_OPTIONALITY(pList)
```

3.4. Az MPM „fő" algoritmus: a maximális minták kinyerése

Az eljárás először az eseménynaplóban tárolt nyomokból kiküszöböli a zajt. A megmaradó nyomokhoz mintát szerkeszt az ismétlődések vizsgálatával. A keletkező mintákból eltávolítja azokat, amelyek részmintái valamely másik mintának (nem maximálisak). A fennmaradt mintákban beazonosítja a párhuzamosságokat és elvégzi az opcionalitás vizsgálatát, melynek eredményeképpen egyetlen minta keletkezik, ami „magában foglalja" a többit. Végül ehhez a mintához elkészít egy automatát, amellyel visszatér és tovább folytatja a nyomok vizsgálatát.

Megjegyezzük, hogy itt a „thresh" kifejezés a „treshvalue" (küszöbszám) rövidítése.

```
1.  MPM(eventlog, thresh)
2.          traces = REMOVE_NOISE(eventlog)
3.          patterns = {}
4.
5.          FOR each trace in traces
6.                  patterns. Add(SOLVE_LOOP(trace))
7.
8.          FOR each pattern p in patterns
9.                  FOR each pattern q in patterns
10.                         IF p is a subpattern of q
11.                                 patterns.Remove(p)
12.                         IF q is a subpattern of p
13.                                 patterns.Remove(q)
14.
15.         FOR each pattern p in patterns
16.                 FOR each pattern q in patterns
17.                         r = SOLVE_CONCURRENCY(p, q)
18.                         patterns. Remove(p)
19.                         patterns. Remove(q)
20.                         patterns. Add(r)
21.
22.         result = SOLVE_OPTIONALITY(patterns)
23.         RETURN CREATE_NFA(result)
```

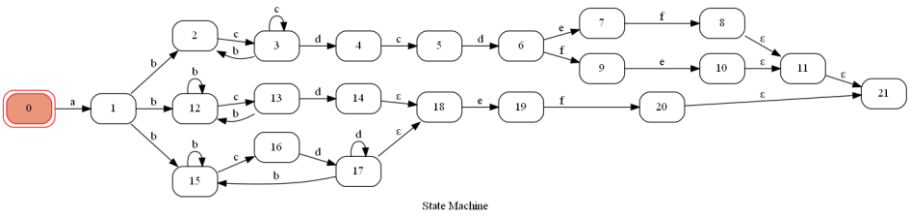
3.5. A kibányászott maximális sémákhoz tartozó gráf megrajzolása

Miután az MPM-eljárás a tevékenységnaplóban megadott nyomokból kiindulva meghatározza a maximális mintákat és megállapítja a párhuzamosságokat és elágazásokat, a keretprogram meghívja a kirajzoló programot. A kirajzoltatás a Pytransitions könyvtárban található GraphMachine osztály által létrehozott gráf draw függvényével történik. Az ábra elkészítésére a „graphviz" vagy a „pygraphviz" Python-alkönyvtárat használja. Az MPM-algoritmushoz tartozó összefogó kódrész a kirajzoltatással együtt a következő:

```
traces = [[c for c in entry] for entry in eventlog]
traces = remove_noise(traces, 0.2)
patterns = list(map(lambda trace: solve_loop(trace), traces))
maximal_patterns = determine_maximal_patterns(patterns, traces)
with_concurrency = maximal_patterns.copy()

i = 0
while i < len(with_concurrency)-1:
    j = i + 1
```

```
    while j < len(with_concurrency):
        r = solve_concurrency(with_concurrency[i], with_concurrency[j])
        if len(r) == 1:
            with_concurrency[i] = r[0]
            del with_concurrency[j]
        else
            j += 1
    i += 1
result = solve_optionality(with_concurrency)
final_nfa = NFA(result.elements)
final_nfa.get_graph().draw("result.png", prog = "dot")
```

Például a $T = \langle abbcbbbbcbbcbbbcbcdef,\ abcbcbccdcdef,\ abcbcbccdcdfe,\ abbbcdbcdddef \rangle$ eseménynaplóhoz a következő automatát készíti el az algoritmus.



3.6. A „keret” algoritmus

Az MPM-eljárásnak a kapott mintákhoz egy olyan automatát kell konstruálnia, ami akceptál minden olyan szót, ami illeszkedik a mintára. Mivel a minta tartalmazhat (egyedi) eseményeket, ciklusokat, párhuzamosságokat, opcionalitást, valamint ezeket a részleteket egymásba ágyazott formában is, így a kidolgozott eljárásnak ezt rekurzívan kell kezelnie. Az itt ismertetett algoritmusban az állapotokat nemnegatív egész számok jelölik, a kezdőállapot 0. Mivel az algoritmus rekurzív, így folyamatosan figyeli, hogy a következő szakasz elejéhez tartozó állapot az előző szakasz végén melyik elemből fog kiindulni (last paraméter). A ciklusnál gondoskodni kell róla, hogy az elem utolsó állapotából vezessen az elsőbe is tranzíció, ezzel biztosítva az ismétlődés lehetőségét. Párhuzamos eseményeknél az automata elágazik többfelé, mindegyiknél az eseménysorozat valamely permutációjához tartozó tranzíciók szerepelnek. Opcionalitás esetén szintén elágazás történik, ahol egy-egy elágazás egy-egy opcionális karaktersorozatnak felel meg. Az eljárás pszeudokódja az alábbi:

```
1.  CREATE NFA (PATTERN)
2.          Q={0}
3.          Σ={e | e  is an event-type in pattern}
4.          T=ø
5.          f=GET_TRANSITIONS_FROM_PATTERN(pattern, -1, Q, T)
6.      F={f}
7.      RETURN(Q, Σ , T, 0, f)
8.
9.
10. GET_TRANSITIONS_FROM_PATTERNS(pattern, prev, Q, T)
11.         IF prev < 0
12.             last = max(Q)
13.     ELSE
14.                     last = prev
15.     FOR element IN pattern
16.         FOR start > end
17.                     IF element is single event
18.                 current = max(Q)+1
19.                         Q = Q ∪ {current}
20.                     last = current
21.                     ELSE IF element is loop
22.                             start =  max(Q)+1
23.             GET_TRANSITION_FROM_PATTERN(loop.elements, last, Q, T)
24.                         end = max(Q)
25.                             e = first event of loop
26.                 T = T ∪ {(e, end, start)}
27.                     last = end
28.                     ELSE IF element is concurent
29.                             ends  = ø
30.             FOR p IN permutations of concurent.elements
31.                 ends = ends ∪ PATTERN GET_TRANSITIONS_FROM_PAT-
32. TERN(p, last, Q, T)
33.             recombine = max(Q)+1
34.             FOR end IN ends
35.                 T = T ∪ {(Ɛ, end, recombine)}
36.             last = recombine
37.         ELSE IF element is optionaly
38.              ends  = ø
39.         FOR eList IN optionaliy.elements
40.              ends = ends ∪ PATTERN GET_TRANSITIONS_FROM_PAT-
41. TERN(eList, last, Q, T)
42.         recombine = max(Q)+1
43.         FOR end IN ends
44.              T = T ∪ {(Ɛ, end, recombine)}
45.          last = recombine
    RETURN last
```

3.7. Elvégzett ellenőrzések, kísérletek

Az előző belső kutatási jelentésekben szereplő javaslatok, valamint az ERPA-csoport tagjainak a javaslatai alapján hozzáláttunk egy ún. tanítóhalmaz összeállításához.

Az alábbi, általam kreált tevékenységnapló estén végeztük el a program részeinek és egésze futásának az ellenőrzését:

T:= {(a, b, b, c, b, b, b, b, c, b, b, c, b, b, b, c, b, c, d, e,f), (a, b, c, b, c, b, c, c, d, c, d, e, f),
(a, b, c, b, c, b, c, c, d, c, d, f, e), (a, b, b, b, c, d, b, c, d, d, d, e, f), (b, c, d, a, a, b, f, g), (b, c, a, a, a, d, b, f, g), (a, b, c, b, c, d, g, f), (a, b, c, b, c, b, c, d, f,), (a, b, c, d, c, d, f, g, g, g), (a, b, c, d, c, d, c, d, f, g, g,, k, l),
(d, a, a, b, c, d, g, f), (c, e, f, g, d, b, b, a, a)}

– ebből a 7. és 8. minta szándékosan bevitt zaj volt.

Ezenkívül, mintegy 100 nyomot tartalmazó tevékenységnaplót állítottunk elő a csoport többi tagja által is használt Process Discovery Contest (2020) mintafilejainak a felhasználásával. Elérési linkje:
https://data.4tu.nl/articles/dataset/Process_Discovery_Contest_2020/14626020

Egyelőre csak a hurkok felismerését, a zaj kiszűrését és a maximális sémák kinyerését vizsgáltuk. Egészen jó (90% feletti) Recall értékeket kaptunk. Hibákat csak a többszörösen összetett hurkok esetén találtunk, illetve a zaj felismerésénél. Ezeket a hiányosságokat szeretnénk a jövőben korrigálni.

3.8. Optimális megoldások keresése egy tanítóhalmazban

Ez tulajdonképpen csak a kérések (requestek) ismeretében megvalósítható és a következő két lehetőség érhető el az algoritmusunkkal: A kérésekkel címkézett minták közül minden kérés esetén kiválaszthatjuk a leggyakoribb mintát(kat), illetve a legrövidebb tranzíciós útvonalat tartalmazó mintát. Ez utóbbi esetén a legrövidebb útvonal alatt egy kezdőállapotból a megoldásba (végállapotba) vezető tranzíciós útvonalat értjük – a fenti algoritmus ezek kinyerésére is alkalmas.

4. Összegzés

A kutatás egy fontos szakasza lezárult az MPM-alapprogram implementálásával. A következő nagyobb szakasz ennek a tesztelésével és tökéletesítésével kapcsolatos.

a) Ezen belül, következő lépésként egy adekvát tanítóhalmaz kialakítása lenne a cél, oly módon, hogy az ebben szereplő mintákat különböző kéréseknek feleltessük meg.
b) Ezekkel először az algoritmus hatékonyságát tesztelnénk a fentiekben már részletezett egyéb mérőszámokat és a korábban már megfogalmazott validációs alapelveket is felhasználva. Ezeket ezután egy külön szubrutin tartalmazná, amit az alapprogramhoz kapcsolnánk.
c) A kapott futtatási eredmények birtokában később javaslatokat fogalmaznánk meg az alapalgoritmus tökéletesítésére. Így például a mostani ellenőrzés fényében szükséges lenne átnéznünk az összetett hurkok képzését, valamint a zaj felismerésének a hatékonyságát.
d) A megkezdett ellenőrzést a párhuzamosságok és opcionalitások feltárásának a vizsgálatával, a folyamatgráf egyszerűsíthető voltának az ellenőrzésével folytatnunk szükséges.
e) Felmerült annak a lehetősége is, hogy az algoritmus kibővítsük NN-alapú részalgoritmusokkal, amelyek a hurkok ismétlődési számát, illetve a párhuzamosított részszekvenciák méretét szabályoznák, oly módon, hogy azok minél „valóságosabbak" legyenek.

Felhasznált irodalom

[1] van der Aalst, W. M. P.: Process Mining: Overview and Opportunities. *ACM Transactions on Management Information Systems*, 2012, vol. 3, no. 2, article 7.

[2] Agrawal, R., Gunopulos, D., Leymann, F.: Mining process models from workflow logs. *Proceedings of the 6th International Conference on Extending Database Technology* (EDBT'98), 1998, LNCS 1377, pp. 469–483.

[3] Cook, J., Wolf, A.: Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology,* 1998 (7), pp. 215–249.

[4] Datta, A.: Automating the discovery of AS-IS business process models: probabilistic and algorithmic approaches. *Information Systems Research*, 1998, vol. 9, pp. 275–301.

[5] Mannila, H., Meek, C.: Global partial orders from sequential data. *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '00), 2000, pp. 161–168.

[6] van der Aalst, W. M. P., Weijters, A. J. M. M., Maruster, L.: Workflow mining: discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 2004, vol. 16, pp. 1128–1142.

[7] Alves de Medeiros, A. K., van Dongen, B. F., van der Aalst, W. M. P. and Weijters, A.J.M.M.: Process Mining: Extending the Alpha-Algorithm to Mine Short Loops. *BETA Working Paper Series*, TU Eindhoven, 2004, vol. 113.

[8] Wen, L., van der Aalst, W. M. P., Wang, J., Sun, J.: Mining process models with non-free-choice constructs. *Data Mining and Knowledge Discovery*, 2007 (15), pp. 145–180.

[9] Wen, L., Wang, J., van der Aalst, W. M. P., Huang, B., Sun, J.: A novel approach for process mining based on event types. *Journal of Intelligent Information Systems*, 2009, vol. 32, pp. 163–190.

[10] Weijters, A. J. M. M., van der Aalst, W. M. P., Alves de Medeiros, A. K.: Process Mining with the Heuristics Miner algorithm. *BETA Working Paper Series*, 2006, TU Eindhoven, vol. 166.

[11] Graves, A. (2013). Generating sequences with recurrent neural network. *arXiv preprint arXiv:1308.0850*.

[12] Folino, F., Greco, G., Guzzo, A., Pontieri, L.: Discovering expressive process models from noised log data. *Proceedings of the 2009 International Database Engineering & Applications Symposium*, 2009, ACM, pp. 162–172.

[13] Islam, M. S., Mousumi, S. S., Abujar, S. and Hossain, S. A. (2019). Sequence-to-sequence Bangla sentence generation with LSTM recurrent neural networks. *Procedia Computer Science*, 152, pp. 51–58.

[14] Ferreira, H., Ferreira, D.: An integrated life cycle for workflow management based on learning and planning. *International Journal of Cooperative Information Systems*, 2006, vol. 15, pp. 485–505.

[15] Burattin, A. and Sperduti, A. (2010, September). PLG: a framework for the generation of business process models and their execution logs. In *International Conference on Business Process Management*, pp. 214–219, Springer, Berlin, Heidelberg.

[16] Liesaputra, V., Yongchareon, S. and Chaisiri, S.: (2016, Sept.) Efficient process model discovery using maximal pattern mining. In *International Conference on Business Process Management*, pp. 441–456, Springer, Cham.

[17] Alves de Medeiros, A. K., Weijters, A. J. M. M., van der Aalst, W. M. P.: Genetic process mining: an experimental evaluation. *Data Mining and Knowledge Discovery,* 2007, vol. 14, pp. 245–304.

[18] Sutskever, I., Martens, J. and Hinton, G. E.: Generating text with recurrent neural networks. *ICML* (2011, Jan.)

[19] Kuo, C. Y., & Chien, J. T. (2018, September). Markov recurrent neural networks. In *2018 IEEE 28th International Workshop on Machine Learning for Signal Processing (MLSP)* (pp. 1–6). IEEE.

[20] Hanga, K. M., Kovalchuk, Y. and Gaber, M. M. (2020). A Graph-Based Approach to Interpreting Recurrent Neural Networks in Process Mining. *IEEE Access*, 8, pp. 172923–172938.

[21] Yunhao, T. Agrawal, S. and Faenza. S. Reinforcement learning for integer programming: Learning to "cut". *International Conference on Machine Learning*, PMLR, 2020.

[22] Agrawal, S. IEOR 8100: Reinforcement learning lectures.

[23] Ritter, G. X., and Urcid, G. (2021). Introduction to Lattice Algebra: With Applications in AI, Pattern Recognition, Image Analysis, and Biomimetic Neural Networks.

[24] Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. *Proc. 8th ACM Intern. Conf. Knowl. Discov. Data Mining*, ACM (2002), pp. 429–435.

[25] Fournier-Viger, P., Lin, J. C. W., Kiran, R. U., Koh, Y. S., & Thomas, R. (2017). A survey of sequential pattern mining. *Data Science and Pattern Recognition*, 1 (1), pp. 54–77.

[26] Pelleg, Dan; Moore, Andrew (1999). Accelerating exact k-means algorithms with geometric reasoning. *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '99*. San Diego, California, United States: ACM Press: pp. 277–281.

[27] Szilágyi Judit et al. (2014). A Support Vector Machine osztályozó eljárás alkalmazása felszínborítás vizsgálatok esetében. *Agriculture Informatics*, p. 46.

[28] Powers, David M W (2011). Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation (PDF). *Journal of Machine Learning Technologies*, 2 (1), pp. 37–63. Archived from the original (PDF) on 2019-11-14.

[29] Deza, Michel Marie, Laurent, Monique (1997). Geometry of Cuts and Metrics. *Algorithms and Combinatorics*, 15, Springer-Verlag, Berlin, p. 27. doi:10.1007/978-3-642-04295-9

MLP-ALAPÚ ELEMIESEMÉNY-ELŐREJELZÉS

MILEFF PÉTER

A kutatás arra a kérdésre keresi a választ, hogy miként lehetséges egy már létező, adott eseményhalmaz alapján olyan neurális hálózatot készíteni, amely a megfelelő szintű betanulás után alkalmas az úgynevezett eseménypredikcióra. A predikció során egy adott eseménysor következő, várható elemét szeretnénk megjósolni. A kutatás ezen részében kizárólag elemiesemény-predikció lehetséges megvalósításának vizsgálata a cél, amely során az MLP- (többrétegű perceptron hálózat) hálózatokkal foglalkozunk részleteiben, a prognosztizálást végző neurális hálózatot MLP-alapokon készítjük el.

1. A kutatás célja és lépései

A mai modern információs rendszerek bonyolult környezetben működnek. A felhasználói és ügyviteli folyamatok évről évre bonyolultabbak lesznek, melyek megfelelő informatikai támogatás nélkül nem bonyolíthatók le hatékonyan. Az irodalomban számos publikáció és tanulmány foglalkozik a felhasználói aktivitások naplózásával, amely során valamilyen formátumú naplófájl létrehozása a cél. A napló (szerencsésebb esetben) tartalmazza a folyamatok elvégzésekor megjelenő elemi eseményeket, azok minden legfontosabb adataival. Persze sok függ attól, hogy a rendszert eleve úgy tervezték, hogy megfelelő minőségben naplózzon, vagy pedig a naplófunkciót csak később „ültették" rá a folyamatokra.

A komplex folyamatok egyik hatékonyságnövelési lehetősége, ha bizonyos részeit vagy akár az egészet automatizálni tudjuk. A feldolgozás felgyorsul, a rendszer szűk keresztmetszetei felderíthetők és megszüntethetők. Az ERPA kutatási projekt egyik fontos alappilléreként szintén megjelenik ez az igény. A naplófájlok feldolgozásával és a modern mesterséges intelligencia eszközeivel vélhetően készíthető olyan szoftver, amely képes a rendelkezésre álló inputsor (az aktuális folyamat első megvalósult lépései) alapján megbecsülni a vélhető következő felhasználói tevékenységet. Az események predikciójára több módszert is kidolgoztak már az irodalomban, jelen munkában az egyik legnépszerűbb eljárást, a neurális hálózatokkal való modellezést szeretnénk részletesebben megvizsgálni.

A kutatási munka célja egy alapvető megvalósíthatósági elemzés elvégzése, amely arra irányul, hogy MLP alapú hálózat segítségével miként lehetséges az események predikciójának megvalósítása. A predikció klasszikus MLP, nem pedig LSTM (Long short-term memory) megoldással való elkészítése több szempontból indokolt:

- Az MLP-alapú hálózatok alkalmazása széles körben elterjedt különféle folyamatokban. Magától értetődik, hogy a kutatás egészéből, mint lehetséges alternatíva nem maradhat ki.

- A komplex eseményeken alapuló későbbi, fejlettebb/bonyolultabb megoldás elkészítésében fontos első lépcsőfok lehet az adatstruktúra és egyéb részek megértésében.
- Az elkészült MLP-modell jó összehasonlíthatósági alapot nyújt a jövőben elkészülő LSTM-hálózathoz.

A kutatás fontosabb lépései a következők:

- Az MLP-alapú neurális hálózatok elméleti hátterének feltárása és megértése.
- Python- és Keras-környezetek megismerése: MLP-hálózatok létrehozása, parametrizálása és a tanítás folyamata.
- Atomi események előrejelzését megvalósító kezdeti MLP-hálózat tervezése és modellezése Python-/Keras-környezetben. Egy egyszerűbb kezdeti prototípus.
- XES-adatstruktúra részletes elemzése, eseménysor átkonvertálása tanítóhalmazra.
- XES-formátum alapján működő prototípusmodell elkészítése Python- és Keras-környezetben.
- Tesztek készítése és elvégzése különböző paraméterbeállítások mellett.
- Eredmények előzetes értékelése, bemutatása.

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

A felhasználói aktivitások figyelésére általában valamilyen aktivitásfigyelő szoftvert alkalmazunk. Az aktivitásfigyelő szoftver rögzíti az alkalmazások és programok használatát a felügyelt munkaállomáson. A képernyőn megjelenő felhasználói tevékenységek egy előre kidolgozott és jól strukturált naplóba kerülnek. A naplók tehát információs adatbázisok, amelyek minden olyan tevékenységet tárolnak, amelyek aznap történtek. A mai modern technológiának köszönhetően számos lehetőség, megoldás áll rendelkezésre a monitorozásra, a tevékenységek figyelemmel kísérésére és kezelésére.

Az elvégzett kutatómunka több nagyobb részre, lépésre bontható, melyet a dokumentum részletesen bemutat. Röviden bemutatásra kerülnek az MLP-hálózatok, majd az MLP-modell alkotás folyamatát Python- és Keras-környezetekben először egy egyszerűbb előrejelzési példán mutatjuk be. Végül egy komplexebb eseménypredikció kerül bemutatásra, amely már a szabványos XES-formátumból dolgozik.

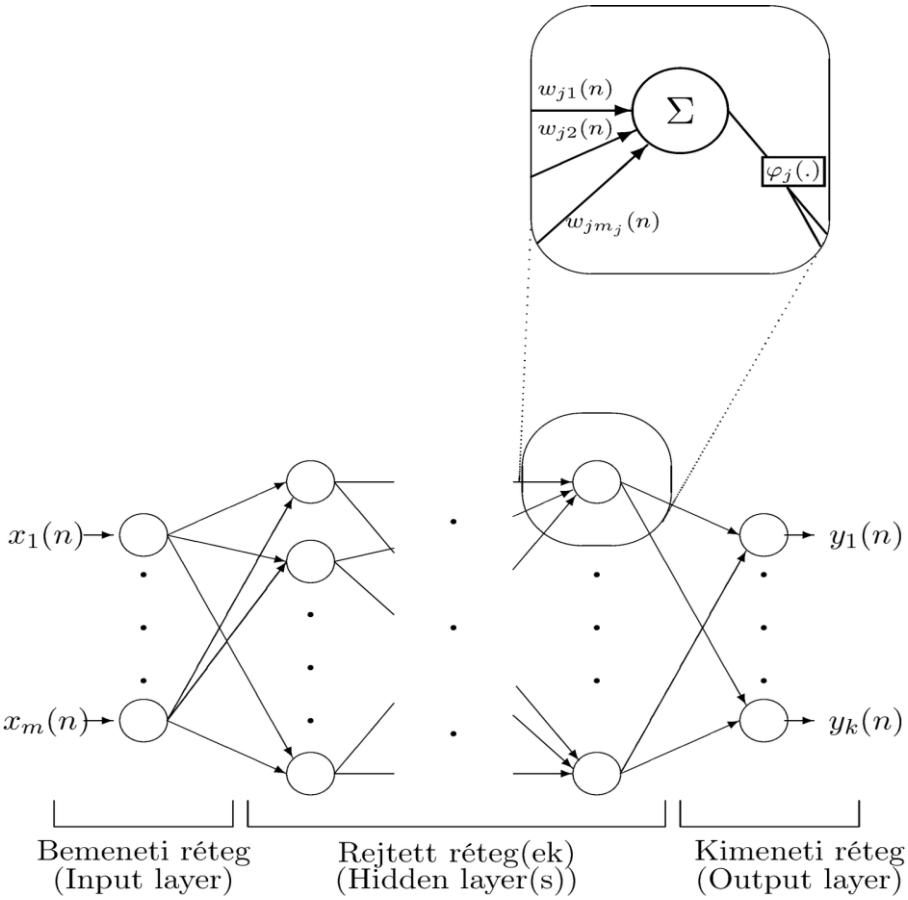
2.1.1. A többrétegű perceptron hálózat

Neurális hálózatokat széles körben alkalmaznak tudományos és műszaki feladatok megoldására. Többek között karakterfelismerésre, képfeldolgozásra, jelfeldolgozásra, adatbányászatra, bioinformatikai problémákra, méréstechnikai és szabályozástechnikai feladatokra használnak különböző neurális hálózatokat. Meg lehet velük oldani olyan összetett problémákat, amelyek visszavezethetők két alapvető feladatra: osztályozásra (azaz szeparálásra) és függvényközelítésre (azaz regressziószámításra). A mesterséges neurális hálózat az idegrendszer felépítése és működése analógiájára kialakított számítási mechanizmus. A neurális hálózatok közismert kezdeti típusa az egyetlen neuronból (idegsejtből) álló perceptron volt. A *Rosenblatt–Novikoff*-féle perceptron konvergencia tétel állítása alapján a perceptron képes elválasztani két lineárisan szeparálható halmazt. A következő lépés az *Adaline* megalkotása volt. Ez úgy tekinthető, mint egy lineáris függvény illesztésére alkalmas eszköz. Ennek tanítása a *Widrow–Hoff-algoritmus*, más néven a *Least mean square* eljárás. Kiderült azonban, hogy több neuront egy rétegbe rendezve sem oldható meg lineárisnál bonyolultabb feladat (Minsky és Papert, 1972). Bonyolultabb elrendezést pedig nem tudtak betanítani. Áttörést a többrétegű perceptron (Multi Layer Perceptron, MLP) tanítására szolgáló eljárás és a hiba visszaáramoltatása (hiba visszaterjesztése, error back-propagation) felfedezése hozta (Rumelhart, Hinton, Williams – 1986). Azóta a neurális hálózatok elmélete és alkalmazásai hatalmas fejlődésen mentek keresztül.

A többrétegű perceptron (multi-layer perceptron, MLP) a gyakorlati feladatok megoldásánál talán a leggyakrabban alkalmazott hálózatarchitektúra. Az MLP egy előrecsatolt neurális hálózat, amely rétegekbe szervezett neuronokból áll, ahol annak biztosítására, hogy a hálózat kimenete a súlyok folytonos, differenciálható függvénye legyen, a neuronok differenciálható kimeneti nemlinearitással rendelkeznek.

A hálózatban háromféle réteget (layer) különböztetünk meg: ***bemeneti, rejtett***, valamint ***kimeneti réteg***ből. A rétegek angol nevei: *input layer, hidden layer, output layer*. Rejtett rétegből tetszőleges számú lehet, viszont bemenetiből és kimenetiből csak egy-egy. A rejtett réteg hozzáadásának az az előnye, hogy kiterjeszti a háló által reprezentálható hipotézisek terét, ez fogja definiálni a hálózat mélységét. Gondoljunk minden egyes rejtett neuronra úgy, mint ami egy lágy küszöbfüggvényt reprezentál a bemeneti térben.

Az alábbi ábra a többrétegű perceptron általános hálózatát mutatja be.

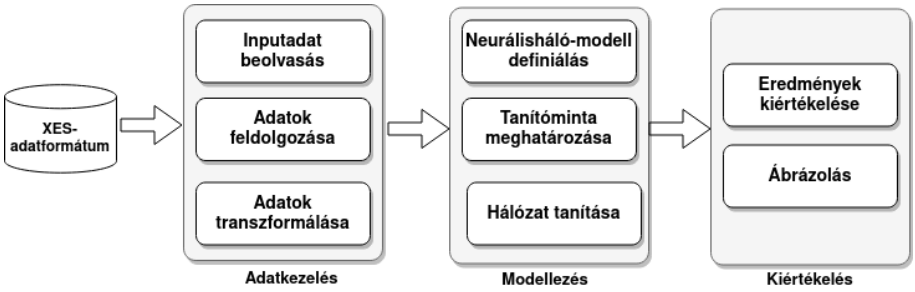


1. ábra. *Többrétegű perceptron hálózat általános modellje*

Bal oldalon van a bemeneti réteg, jobb oldalon a kimeneti, közöttük pedig egy vagy több rejtett réteg. A jel balról jobbra áramlik, azaz egy adott rétegbeli neuron bemenete (inputja) a tőle balra lévő rétegbeli neuronok kimenete (outputja). Az általunk tárgyalt modell esetén nincs kapcsolat rétegen belül és távolabbi rétegek között sem. Viszont minden neuron kapcsolatban van a vele közvetlenül szomszédos rétegek minden neuronjával. A többrétegű perceptron fontos tulajdonsága, hogy minden neuronjának saját aktivációs függvénye és saját súlyai vannak.

A rejtett neuronok számának előzetes meghatározása még napjainkban sem jól megoldott probléma. Számos kutatás foglalkozik az egzakt meghatározásával, vagy valamilyen megfelelő becslés adásával, de a gyakorlatban jelenleg empirikus alapokon történik a rejtett neuronok számának a meghatározása.

2.1.2. A többrétegű perceptron hálózat tanítása



2. ábra. *A technológiai folyamat lépései*

A többrétegű perceptron egy előrecsatolt hálózat (feedforward network). Azaz az inputjel rétegről rétegre halad előre, az outputréteg pedig megadja a kimenő jelet. A többrétegű perceptron tanításánál nehézséget jelent, hogy egy neuron kimenetét nem mindig tudjuk közvetlenül minősíteni, hiszen csak a kimeneti réteg esetében van elvárásunk a kimenetre, így csak ezeknél a neuronoknál tudjuk közvetlenül értelmezni a hibát. Erre jelent megoldást a hiba-visszaterjesztéses módszer, amely kihasználja, hogy egy adott neuron bemenete az előző réteg kimenete így a rétegeken visszafelé haladva tudja minősíteni a neuronok kimenetét és meghatározni a korrekciót.

A tanítás fő lépései:

- Megadjuk a kezdeti súlyokat.
- A bemeneti jelet (azaz a tanítópontot) végigáramoltatjuk a hálózaton, de a súlyokat nem változtatjuk meg.
- Az így kapott kimeneti jelet összevetjük a tényleges kimeneti jellel.
- A hibát visszaáramoltatjuk a hálózaton, a súlyokat pedig megváltoztatjuk a hiba csökkentése érdekében.

A többrétegű perceptron tanítása a *hiba visszaáramoltatása* módszerrel (hiba visszaterjesztése, error/back-propagation algorithm) történik.

2.1.3. A többrétegű perceptron hálózat a gyakorlatban

A gyakorlatban a többrétegű perceptron hálózatok széleskörűen alkalmazhatók számos különböző problémák megoldásában. Egyik leggyakoribb alkalmazási és a projekt szempontjából releváns területe az úgyneverett idősorok / atomi eseménysorok várható jövőbeli értékeinek előrejelzése. Az eseménysorok várható következő elemének predikciója kihívást jelent, különösen akkor, ha hosszú sorozatokkal, zajos adatokkal, többlépcsős előrejelzésekkel és több bemeneti és kimeneti változóval kell dolgozni. A mesterséges neurális hálózatok, leginkább a mély tanulási módszerek ígéretes lehetőséget kínálnak az idősorok előrejelzésére, mint például az időbeli függőség automatikus megtanulása és az időbeli struktúrák, például a trendek és a szezonalitás automatikus kezelése.

2.1.3.1. Adatstruktúra-előkészítés

Ahhoz, hogy „bármilyen" jellegű problémát modellezni lehessen egy neurális hálózattal, ahhoz az adatok megfelelő előkészítése, a megfelelő struktúra kialakítása elengedhetetlen. A gyakorlati gépi tanulás többsége felügyelt tanulást használ. Jelen fejezetben az idősorok adatait alakítjuk át úgy, hogy az alkalmas legyen a felügyelt tanulási formátumnak.

A felügyelt tanulás az, ahol van bemeneti változó (X) és kimeneti változó (y), és egy algoritmus segítségével megtanulja a leképezési funkciót a bemenetről a kimenetre. A cél az, hogy olyan jól közelítsük meg a valódi mögöttes leképezést, hogy új bemeneti adatok birtokában megjósolhassuk az adatok kimeneti változóit.

Ha adott egy adathalmaz-sorozat, akkor az adatokat átalakíthatjuk úgy, hogy azok felügyelt tanulási problémának minősüljenek. Ezt úgy tehetjük meg, hogy a korábbi lépéseket használunk bemeneti változóként, és a következő lépéseket használjuk kimeneti változóként.

Példaadatsor: 1, 2, 3, 4, 5, ...

A sorozat bemeneti és kimeneti komponensekkel rendelkező olyan mintákká alakítható, amely a betanítási folyamat részeként felhasználható, például egy mély tanulási neurális hálózat oktatására.

X,	**y**
[1, 2, 3]	4
[2, 3, 4]	5

Ezt *csúszóablak-transzformáció*nak nevezik, mivel ez olyan, mint egy ablak elcsúsztatása olyan korábbi megfigyelések között, amelyeket a modell bemeneteként

használnak a sorozat következő értékének előrejelzésére. Ebben az esetben az ablak szélessége 3 időlépés.

Az MLP-modellek esetében a Keras ebben formátumban fogja várni az adatokat, ezért a modellalkotást mindig meg fogja előzni egy adat-előkészítés, transzformáció, amely a fenti vagy ahhoz hasonló formátumra alakítja az adatokat.

2.1.3.2. Egyváltozós MLP-modell

A mesterséges neurális hálózat építését egy egyszerű, úgynevezett egyváltozós eseménysoron alapuló modell építésével kezdjük. Az egyváltozós eseménysorok olyan adathalmazok, amelyek egyetlen megfigyelési sorozatból állnak időbeli sorrendben. Egy olyan modell kialakítására van szükség, amely képes lesz a korábbi megfigyelésekből fakadó sorozatok tanulására, majd pedig a sorozat következő értékének előrejelzéséhez. A következőkben ezt mutatjuk be részleteiben.

2.1.3.2.1. Adatok előkészítése

Az egyváltozós sorozat modellezése előtt az adatokat szintén a megfelelő módon elő kell készíteni, ahogyan már a korábbiakban említésre került. Az MLP-modell olyan tanulást fog végezni, amely során a múltbéli eseményeket, mint egy bemeneti értékek sorozatát leképezi kimeneti eseménnyé. Ahhoz, hogy ez megvalósulhasson, a megfigyelések sorozatát több tanulómintára kell bontani, amelyekből a modell tanulhat.

Tekintsük az alábbi egyváltozós sorozatot mintaként:

[10, 20, 30, 40, 50, 60, 70, 80, 90]

Ahhoz, hogy tanuló mintát hozzunk létre, a szekvenciát több bemeneti/kimeneti adatra kell szétosztani, amelyeket *minták*nak nevezünk. Három „időlépést”, azaz három sorozati értéket használunk bemenetként, és egy időlépést használunk kimenetként a tanuló egylépéses előrejelzéshez.

X,	y
10, 20, 30	40
20, 30, 40	50
30, 40, 50	60
...	

A sorozatból jól látszik, hogy két logikai egységet képezünk: egy adott inputmintának milyen kimenete lesz. Azaz jelen esetben mely sorozati értékek után minek

kell következnie. Az adatok egységessége kulcsfontosságú, különben a program nem lesz képes feldolgozni azokat.

Az adatok ilyen alakra való hozását valamilyen automatikus megoldással célszerű elvégezni. Az alábbi mintapélda és a benne szereplő *split_input_sequence*() függvény megvalósítja ezt a viselkedést, és egy adott egyváltozós sorozatot több mintára oszt fel, ahol minden minta meghatározott számú időlépéssel rendelkezik, és a kimenet egyetlen időlépés. A megvalósító Python-kód:

```
from numpy import array
# split a univariate sequence into samples
def split_input_sequence(sequence, n_steps):
	X, y = list(), list()
	for i in range(len(sequence)):
		# find the end of this pattern
		end_ix = i + n_steps
		# check if we are beyond the sequence
		if end_ix > len(sequence)-1:
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
		X.append(seq_x)
		y.append(seq_y)
	return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_input_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
	print(X[i], y[i])
```

A mintapélda futtatása során az egyváltozós sorozat hat mintára oszlik, ahol minden minta három bemeneti és egy kimeneti időlépéssel rendelkezik.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

2.1.3.2.2. MLP-modell-idősorok előrejelzéséhez

Az idősorok becsléséhez szükséges Keras-környezetben megvalósított kezdeti MLP-modell a következő. Jelen mintakód nem tartalmazza a *split_input_sequence*() függvényt, de a gyakorlatban annak szerves része kell legyen.

```
# univariate mlp example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_input_sequence(raw_seq, n_steps)
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

A modell elemeinek részletes magyarázata

A Keras-könyvtár központi eleme a modell, amelyet a könyvtárban *(keras.models)* a ***Sequential*** osztály reprezentál. Ez a modell egy alap a Keras-csomagban, működését tekintve a modell rétegek lineáris halmazaként valósul meg.

Egy ***Sequential*** modellt többféleképpen hozhatunk létre, többféleképpen adhatjuk meg a rétegeket. Legegyszerűbb megadás a konstruktorban való definiálás:

```
from keras.models import Sequential
model = Sequential(...)
```

Hasznosabb megoldás azonban az, ha a modell létrehozásakor a szükséges rétegeket a végrehajtani kívánt számítás sorrendjében adjuk meg, például:

```
from keras.models import Sequential
model = Sequential()
model.add(...)
model.add(...)
model.add(...)
```

Modellbemenetek

A modell első rétegében meg kell adni a bemenet alakját. Ezt egy úgynevezett *Dense* típusú réteggel valósítjuk meg.

A *Dense* réteg az alábbi műveletet valósítja meg: output = aktivációs_fgv (dot (input, kernel) + bias), ahol az aktivációs_fgv az elemek szerinti aktiválási függvény, a kernel a réteg által létrehozott súlymátrix, és a bias pedig a réteg által létrehozott egy eltolásvektor.

Egy alap Dense réteg létrehozásához a batch és a bemeneti attribútumok számának megadása szükséges.

Például egy ***Dense*** típusú réteghez, ha 8 bemeneti minta-darabszámot szeretnénk rendelni, akkor az alábbiak szerint adhatjuk meg:

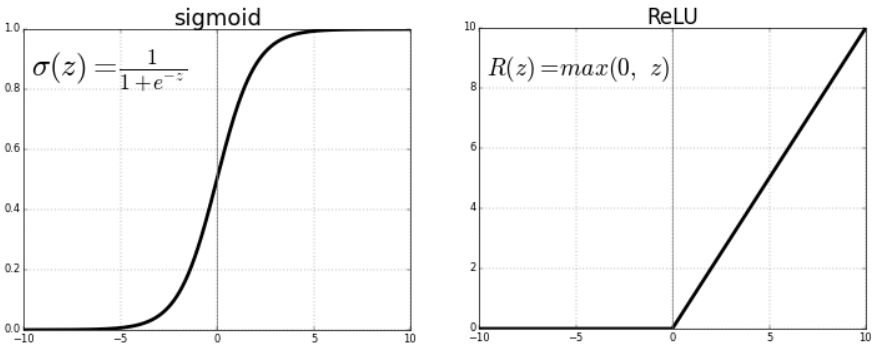
```
Dense(16, input_dim=8)
```

Aktivációs függvény

Az aktivációs függvény a neurális hálók működtetéséhez szükséges elem. A háló úgy működik, hogy a hálóban lévő egyes neuronok kapcsolatban állnak. Az egyes neuronok a bemenetük és a kimenetük között egy transzformációt végeznek. Minden egyes neuronkapcsolathoz tartozik egy numerikus súly. A neuron először veszi a bemeneteinek egy súlyozott összegét, majd ezek után ezt az értéket keresztülvezeti az aktivációs függvényen. A Keras számos szabványos neuronaktivációs függvényt támogat, például: *softmax, relu, rectifier, tanh* és *sigmoid*.

A ***Sigmoid*** és ***ReLU*** a világ leggyakrabban használt aktivációs függvényei. Szinte minden konvolúciós mély tanulási hálózatban használják. A bemutatott MLP-modellben a ReLU-függvény került alkalmazásra.

A modell definíciójában fontos a bemenet alakja, ezt várja a modell bemenetként az egyes mintákhoz az időlépések számát tekintve. Ez az a pont, ahol a létrehozandó modellt és a bemeneti mintasort egymáshoz igazítjuk. Ez a gyakorlatban azt jelenti, hogy *split_sequence()* függvény argumentumaként megadott felbontási számot (n_steps) a modell *input_dim* argumentumának is át kell adni, így a modell által várt adatstruktúra és a *split_sequence()* függvény által készített formátum egyezni fog.



3. ábra. *Sigmoid és ReLU aktivációs függvények.*

A modell definiálása után lehetőségünk van a hálót a tanító-adatsorra betanítani.

```
# fit model
model.fit(X, y, epochs=2000, verbose=0)
```

Miután a hálózat betanult, már használhatjuk előrejelzésre. A bemenet megadásával megjósolhatjuk a sorozat következő értékét. Például ha a bemenet értéke:

[70, 80, 90]

A hálózat által elvárt és jósolt eredmény: [100]

A Kerasban a predikció elvárja, hogy a bemeneti adat kétdimenziós legyen [minták, jellemzők], ezért az előrejelzés előtt át kell alakítani a bemeneti mintát. Erre a *reshape()* Python-függvény ad lehetőséget.

```
demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
```

Megjegyzés: A kód futtatása során az eredmények eltérőek lehetnek, tekintettel az algoritmus vagy az értékelési eljárás sztochasztikus jellegére. Célszerű a példát többször futtatni és összehasonlítani az eredményeket, azok átlagát nézni.

2.1.4. MLP-alapú atomiesemény-predikció

A neurális hálózatok numerikus értékekkel dolgoznak. Ahhoz, hogy egy nem numerikus alapú problémát neurális hálózattal modellezni lehessen, ahhoz numerikus alakra kell leképezni. Az esemény-előrejelzési feladat is természetesen ebbe a csoportba tartozik, ugyanis a gyakorlatban az egymástól elkülönülő eseményeket általában valamilyen atomi azonosítóval modellezik. Míg a gyakorlatban, egy-egy valós működő információs rendszerben a számalapú eseményreprezentáció is elméletben megfelelő lenne, azonban a numerikus reprezentációtól az emberi olvashatóság miatt gyakran eltérnek.

Feltételezhetjük tehát, hogy az elemi eseménysor valamilyen szöveges azonosítóval van ellátva. Az egyszerűség kedvéért vizsgáljuk az alábbi eseménysorpéldát, ahol az egyes elemi események egy betűvel reprezentáltak:

['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

Ahhoz, hogy a neurális hálózat inputadathalmazaként a fenti adatsor megadható legyen, le kell képezni valamilyen numerikus értékké. Jelen egyszerű példában magától adódik az az érték hozzárendelési módszer, ahol az abécé betűin sorban haladva kerülnek az értékek társításra a betűkhöz. Az alábbi Python-mintakód ezt valósítja meg:

```
raw_seq_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
letters_dictionary = {chr(i+96):i for i in range(1,27)}
```

A kódrészlet egy úgynevezett „dictionary”-t készít el, amely minden betűhöz egy számot rendel hozzá a következőképpen:

Az elkészült adatsorból pedig már kialakítható az a megszokott input-adatstruktúra, amit a neurális hálózati modell elvár:

```
raw_seq = []
for char in raw_seq_char:
  raw_seq.append(letters_dictionary[char])
```

Ezzel megkaptuk a számsoralapú inputvektort, amelyet a fenti példába illesztve a hálózat betanítása elvégezhető.

2.1.4.1. MLP-alapú eseménypredikció XES-formátum esetén

A kutatási munka végső célja, hogy olyan neurális hálózati modell kerüljön kialakításra, amely képes XES állományokban szereplő adatok tanulására, majd az események predikciójára. A fejlesztéshez egy szabványos benchmark *pdc_2016_1.xes* adatállomány állt rendelkezésre, a kísérleteket ezen végeztük el.

Jellemzői:

- szimbólumkészlet mérete: 18
- a leghosszabb szekvencia hossza: 30
- adatminták darabszáma: 1000

A kutatási jelentés korábbi részeiben részletesen bemutatásra került az XES-formátum, ezért ebben a dokumentumban nem térünk erre ki. Ahhoz, hogy az XES-adatok az MLP számára kezelhetők legyenek, az adathalmazon beolvasás után számos transzformációt kell elvégezni. A beolvasást és transzformációt végző kódrész a következő:

```
def load_graphs(self, xmlfile="pdc_2016_1.xes"):
    t_lists = []
    f_dict = dict()

    tree = ElementTree.parse(xmlfile)
    root = tree.getroot()

    for item in root.findall('.//trace'):
        t_lists.append([])
        for eve in item.findall('event'):
            for vv in eve.findall('string[@key="concept:name"]'):
                t_lists[-1].append(vv.attrib['value'])

    n = len(t_lists)
    print("N=", n)

    for i in range(n):
        kk = ''.join(t_lists[i])
        if kk in f_dict.keys():
            f_dict[kk] += 1
        else:
            f_dict[kk] = 1

    self.c_dict = dict()
    self.data = []
    i = 0
```

```
    self.max_sequence_length = 0
    for kk in f_dict.keys():
        kl = []
        if len(kk) > self.max_sequence_length:
            self.max_sequence_length = len(kk)
        for c in kk:
            if c not in self.c_dict.keys():
                self.c_dict[c] = len(self.c_dict) + 1
            kl.append(self.c_dict[c])
        i += 1
        self.data.append(kl)

    n2 = len(self.data)
    self.c = len(self.c_dict)
    print("C", self.c, "L", self.max_sequence_length, "N", n2)

    for i in range(n2):
        if len(self.data[i]) < self.max_sequence_length:
            self.data[i] = [0 for _ in range(self.max_sequence_length - len(self.data[i]))] + self.data[i]
```

Az algoritmus célja, hogy a fájlbetöltés után egységes formátumra hozza az egymás után következő összetartozó eseménysorozatot. Mivel az XES-en belül nem minden tevékenység eseményeinek darabszáma azonos, de a Keras-alapú MLP-modell viszont azonos méretű "szeletekben" várja az adatokat, így nem marad más lehetőség, mint az egységes formátumra hozás. Az algoritmus meghatározza a leghosszabb mintát, és ahhoz igazítja a többi tevékenység eseménysorát úgy, hogy a kevesebb darabszámú eseménysorokat balról nullákkal egészíti ki a leghosszabb méretűnek megfelelően. Példa:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 13, 13, 13, 13, 15, 14, 16, 1, 6, 3, 2, 4, 5, 7, 8, 10, 9, 11, 12]
```

A következő fontos elem az MLP-modell leírása. A következő kód ezt mutatja be:

```
def create_mlp_model(self, h):

    model = tf.keras.Sequential(
      [
        tf.keras.layers.Dense(h, activation='relu', input_dim=self.m),
  tf.keras.layers.Dense(h),
        tf.keras.layers.Dense(len(self.c_dict) + 1, activation='softmax'),
      ]
    )
     model.compile(optimizer='adam',loss='categorical_crossentropy',metrics=['accuracy'])
return model
```

Konkrét, 100 darab ***h*** értékkel az alábbi hálózat jön létre a Kerasban:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_2 (Dense)              (None, 100)               1100
_________________________________________________________________
dense_3 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_4 (Dense)              (None, 19)                1919
=================================================================
```

Végül, de nem utolsósorban két fontos függvényt mutatunk be:

```
def create_dataset(self, dataset, look_back, data_x, data_y):
        c = len(self.c_dict)
        for i in range(len(dataset) - look_back - 1):
            if dataset[i + look_back] > 0:
                a = dataset[i:(i + look_back)]
                for j in range(len(a)):
                    a[j] = a[j] / c
                b = [0 for _ in range(c + 1)]
                b[dataset[i + look_back]] = 1
                data_x.append(a)
                data_y.append(b)
        return np.array(data_x), np.array(data_y)

    def gen_train_data(self, m):
        self.m = m
        train_x, train_y = [], []
        for i in range(len(self.data)):
            self.create_dataset(self.data[i], self.m, train_x, train_y)

        return train_x, train_y
```

A *gen_train_data* függvény szerepe, hogy egy megfelelő tanítási adathalmazt hozzon létre. Ehhez a betöltött XES-adatokat használja fel. A paraméterben megkapott érték alapján megadott méretű inputadat-szeletekből álló tömböt hoz létre a *create_dataset* függvény segítségével a korábbiakban bemutatott inputadatmintához hasonlóan.

A fenti függvények egy osztályban kaptak helyet. A működtetést végző kód az alábbi:

```
from MLPHelper import MLPHelper
import numpy as np
import matplotlib.pyplot as plt

if __name__ == "__main__":
    mlp = MLPHelper()
    mlp.load_graphs()
    (train_x, train_y) = mlp.gen_train_data(10)
    model = mlp.create_mlp_model(100)
    model.summary()

    train_x = np.array(train_x)
    train_y = np.array(train_y)

    history = model.fit(train_x, train_y, epochs=200, verbose=1)

    train_mae = history.history['accuracy']
    plt.plot(train_mae, label='train accuracy')
    plt.show()
```

Az adatok tanítását a *model.fit* függvény végzi, majd a tanítás közben mért hatékonyságot egy diagramban ábrázolja.

2.2. Kiértékelések eredményeinek bemutatása

A kutatás ezen fázisában a kiértékelés mérőszámaként a tanulás helyességét tudjuk értelmezni. A Keras a tanítási folyamat során beépített lehetőséget kínál arra, hogy a tanítási folyamat során keletkező ***„loss"*** és ***„accuracy"*** értékek mérhetők legyenek, melyek későbbi összehasonlításokhoz használhatók fel.

A fenti kódsorokban szereplő *„**categorical_crossentropy**"* definiálja a modellben a hibafüggvényt. A *categorical_crossentropy* mérőszámot a többosztályú osztályozás esetén szokás használni.

A tanítási folyamat hosszú és időigényes. A fejlesztő környezet ezt és a legfontosabb értékeket szövegesen mutatja. Példa:

```
Epoch 1/200
341/341 [==============================] - 0s 572us/step - loss: 2.5461 - accuracy: 0.1867
Epoch 2/200
341/341 [==============================] - 0s 791us/step - loss: 1.8668 - accuracy: 0.2925
```

Epoch 3/200
341/341 [==========] - 0s 790us/step - loss: 1.6797 - accuracy: 0.3251
Epoch 4/200
341/341 [==========] - 0s 787us/step - loss: 1.6208 - accuracy: 0.3484
Epoch 5/200
341/341 [==========] - 0s 808us/step - loss: 1.5994 - accuracy: 0.3552
Epoch 6/200
341/341 [==========] - 0s 765us/step - loss: 1.5832 - accuracy: 0.3594
Epoch 7/200
341/341 [==========] - 0s 569us/step - loss: 1.5487 - accuracy: 0.3696
Epoch 8/200
341/341 [==========] - 0s 548us/step - loss: 1.5287 - accuracy: 0.3761
Epoch 9/200
341/341 [==========] - 0s 566us/step - loss: 1.5229 - accuracy: 0.3904

A kísérletben több epoch értékkel végeztünk tanítást. Az epoch szám egy olyan hiperparaméter, amely azt a darabszámot definiálja, hogy hányszor haladjon végig a tanítóalgoritmus a teljes tanítómintán.

A neurális hálózatok nem determinisztikus rendszerek, amelyek ugyanazon inputra akár eltérő outputtal szolgálhatnak. Ebből kifolyólag a kutatási munka során egy-egy eredmény kiértékelését mindig több különböző futtatás alapján határoztuk meg.

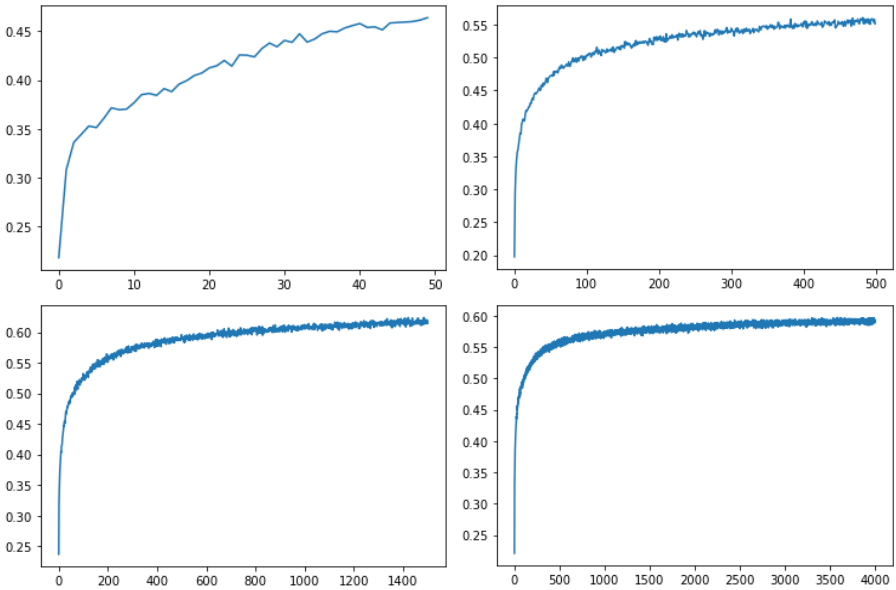
Az alábbi táblázat az epoch értékekhez tartozó loss és accuracy értékeket foglalja össze.

1. táblázat. *Futási eredmények összehasonlítása*

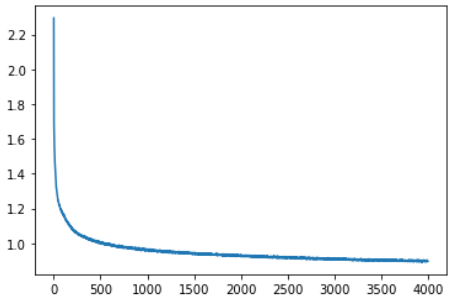
Epoch	Loss	Accuracy	Time
50	1,2645	0,4345	13 sec
200	1,1102	0,51	35 sec
500	1,0078	0,5586	1:55 min
1000	0,9360	0,5874	3:31 min
2000	0,9132	0,6018	7:23 min
4000	0,8642	0,6109	15:33 min

2.3. Eredményeket szemléltető diagramok

Az alábbi diagramok a különböző epoch számmal készített tanulási folyamatok eredményét, az „accuracy” értékének változását szemléltetik.



Veszteség függvény értékének változása 4000 epoch szám esetén:



3. Összegzés

A mesterséges intelligencia területei az utóbbi években nagymértékű fejlődésen mentek keresztül. Nem csupán újabb modellek születtek, hanem a rendelkezésre

álló szoftvereszközök egy sokkal szélesebb spektruma érhető ma már el mindenki számára. A bemutatott példákból jól látszik, hogy a Python- és Keras-környezetek alkalmasak a komolyabb feladatok elvégzésére is. A bemutatott MLP-hálózatok egy jól működő alternatívát kínálnak az események predikciójának megvalósítására. A tesztekben használt mintaadathalmazon megfelelő eredményt lehet elérni a modell alkalmazásával. A jövőben, az LSTM-modell mellett célszerű egy, a problémát megoldó MLP-modell-változatot is fenntartani és kidolgozni, amely egy lehetséges működő alternatívát kínál majd a futtatási eredmények összehasonlítására.

NEURÁLIS HÁLÓ ALAPÚ ÖSSZETETT ESEMÉNYLÁNC FELTÁRÁSA

KOVÁCS LÁSZLÓ

A kutatási modul célja annak bizonyítása, hogy a mintafeltárás neurális háló alapú megközelítéssel is lehetséges legalább olyan hatékonysággal, mint amit a klasszikus, nem háló alapú módszerek biztosítanak. A kutatási modulban olyan neurálisháló-architektúra kerül kidolgozásra, amely

- *minimumelvárásként a művelettípusokat és azok időbeliségét,*
- *maximumelvárásként a műveletek költség- és sikerességstátuszát is*

figyelembe véve képes a folyamatmodell előállítására.
A modulban kidolgozandó modell jellegére minimumelvárás, hogy ne csak az esemény szekvenciákat, hanem az eseményfolyam-gráfmintákat is képes legyen előállítani.

1. A kutatás célja és lépései

Az első lépés a komplex gráfok elemzése volt, annak meghatározása, hogy milyen folyamatmodellezési szabványok vannak és azok mennyire támogatják a komplex eseményeket. Ennek során az alábbi lépéseket végeztem el:

- modellezési nyelvek áttekintése
- komplex események típusainak elemzése
- AND-típus vizsgálata
- XOR-típus vizsgálata
- ciklus vizsgálata
- eseményparaméterezések vizsgálata (ágens, objektum, ...)
- eseménygráf-feltáró

Az időszakban elvégzett kutatás az MLP-alapú eseményfeltáró modell elemzésére irányult, ahol a vizsgálat fő célja annak felderítése, hogy milyen mértékben alkalmas a modell valamely továbbfejlesztése az összetettebb gráfminták feltárására, illetve betanulására.

A tervezés során az alábbi vizsgálati pontokra tértem ki:

- eseménysor konverziója elemi előrejelzési lépésekre
- a háló tanítóhalmazának struktúrájának az előállítása
- tanító halmazok felépítése
- mintarendszer implementálása Pythonban
- tesztek elvégzése különböző paraméterbeállítások mellett
- rétegszám hatásának az elemzése

- inputvektor hosszhatásának az elemzése
- neuronszám hatásának az elemzése
- aktivációfüggvény hatásának az elemzése
- eredmények előzetes értékelése

A kutatás következő pontja a későbbi predikciós motorok tesztelési környezetének az előkészítése volt, amely egy tanítóhalmaz generáló rendszer előállítására szolgált. Ennek során pontosítani kellett a vizsgált modelleket, azok paramétereit, a generálás módját és a kimeneti eredmény formátumot.
A kimenetnek közvetlenül támogatnia kell a neurális hálók bemeneti formátumát.

A tervezés során az alábbi vizsgálati pontokra tértem ki:
- csomóponttípusok azonosítása
- eseményparaméterek azonosítása
- gráfreprezentáció meghatározása
- sémagráf-felépítés algoritmus kidolgozása
- sémafelépítő algoritmus kidolgozása
- sémavalidáció kifejlesztése
- random gráfpéldány-generáló algoritmus kidolgozása
- eredmények megjelenítési módjának megtervezése
- eredmények exportálása
- exportadatok utófeldolgozása

Az időszak további fő feladata a komplex eseménygráf feltárására szolgáló neurálisháló-modell megalkotása volt. Első lépésként két fő alternatíva az LSTM- és MLP-hatékonyság összevetését végeztem el. Ennek során az alábbi lépéseket végeztem el:
- elemi szekvenciák tanítóhalmazának előállítása
- LSTM-háló felépítése
- MLP-háló felépítése
- hatékonysági tesztek elvégzése
- paraméterezés hatásának tesztelése
- tesztek kiértékelése
- NN-típus kiválasztása

Az eredmények alapján az MLP került kiválasztásra az NN-modul elkészítéséhez.

A győztes NN-modell meghatározása után az NN-modell-architektúra részletes kidolgozása volt a fő cél. Ehhez előbb a gráf NN-modell logikáját kellett meghatározni, ennek során az NN-modulok specifikálását és azok kapcsolatának a kiépítését végeztem el.

A tervezés során az alábbi vizsgálati pontokra tértem ki:

- gráf felbontása feldolgozható modulokra
- a modulok előállítási, feltárási algoritmus kidolgozása
- az induló eseménysor modulokra bontási algoritmusának a kidolgozása
- a modulszintű elemzés megtervezése
- a modulok integrációs keretének a kidolgozása

Az időszak soron következő feladata a komplex eseménygráf feltárására szolgáló neurálisháló-modell véglegesítése volt. Kidolgozásra került a tanítási és a predikciós architektúra modell, elkészült a rendszer implementációja. Az elkészült rendszer ellenőrzésére teszteket hajtottam végre a generált tanítóhalmazokra építve.

Ennek során az alábbi lépéseket végeztem el:

- a réteges modell megvalósítása önálló MLP-modulokon keresztül
- az egyes MLP-modulok kapcsolatának kiépítése az adatok szintjén
- MLP-hálók implementálása
- hatékonysági tesztek elvégzése
- paraméterezés hatásának tesztelése
- tesztek kiértékelése
- keretrendszer értékelése

Az elkészített hálómodell tesztelése AND- és XOR-típusú szinkronizációs pontokat tartalmazó gráfmintákkal.

Az elért eredmények bemutatására egy külön publikációt tervezünk, ennek előkészítésére az alábbi lépéseket végeztem el:

- a szerzői csapat összeállítása, a feladatok kiosztása
- háttérirodalom szisztematikus feldolgozása
- a kidolgozott modell egyediségének kiemelése
- a tervezett cikk struktúrájának meghatározása

Az MLP/LSTM összevetési kísérleteket, a tanuló és a predikciós keretrendszer implementációját Keras-/Tensorflow-környezetben hajtottam végre. Majd a második fázisban csak elméleti munka folyt.

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

Az irodalom elemzése alapján megállapítható, hogy az üzleti folyamatokat tevékenységekkel írják le, a tevékenységek sorrendjét pedig alkalmi függőségek modellezik. Ezt a relációt gráffal ábrázolják, amely csomópontokból és irányított élekből épül fel a csomópontok között, jelezve a folyamatok időrendi sorrendjét. Az él okozati és időbeli tulajdonságokat is leírhat, ill. meghatározza az adatok létrehozását és felhasználását a döntések modellezéséhez és annak módját. Az egyik legrégebbi folyamatmodellezési nyelv a Petri-háló, amely rendelkezik néhány magasabb szintű kiterjesztéssel, míg a legkifejezőbb üzleti folyamatmodell és jelölés a BPMN-nyelv [16]. Ezekhez a modellekhez szabványos XML-alapú adatcsereformátumokat fejlesztettek ki. A BPMN szabvány magában foglalja az XML Process Definition Language (XPDL), míg a Petri-hálók automatikusan feldolgozhatóak a Petri Net Markup Language (PNML) eszközzel. A Workflow Patterns Initiative szisztematikus elemzése eredményeként gráfminták gyűjteményét hozták létre. melyet a YAWL nyelv támogat. Ezek a minták az összes munkafolyamat-perspektívát lefedik. Például vannak vezérlési folyamatminták, adatminták vagy erőforrásminták. Vizsgálataink a control-flow perspektívára fókuszálnak, ahol 43 mintázat található, melyeket 8 osztályba sorolták.

A főbb vezérlési elemek:

- szekvencia
- elemi elágazás (XOR-ág)
- több választós elágazás (OR-ág)
- párhuzamos ág (AND-ág)

A folyamatfák blokkstrukturált modelleket képviselnek egy hierarchikus folyamatjelölést alkalmazva, ahol a (belső) csomópontok operátorok, mint pl. sorrend és választás, a levelek pedig tevékenységek. A levél csomópontjai egy process fa az eseménycsomópontoknak felel meg, míg a nem levél csomópontok operátorok az események végrehajtási sorrendjét leíró csomópontok. A modell biztosítja a következő operátor-csomópontokat: eseményszekvencia (→), párhuzamos végrehajtás (AND) (∧), nem kizárólagos választás (OR) (∨), kizárólagos választás (XOR) (×) és eseményhurok.

A fejlesztés során elsőként a gráf absztrakt modelljét kellett kidolgozni. Ennek eredményeképp a sémagráfot és a sémacsomópontokat az alábbi paraméterek jellemzik:

- eseménytípusok száma
- aktorok száma (több ágens, munkás dolgozhat egyidejűleg a rendszerben)
- megmunkálási idő alapparaméterei az egyes eseménytípusoknál
- bemenő objektumok az egyes eseménytípusoknál
- kimenő objektumok az egyes eseménytípusoknál
- eseményrákövetkezési reláció

A sémagráf felépítéséhez egy programkódot kell írni, melyben példányosítjuk a kívánt sémaelemeket.

A keretrendszer grafikusan megjeleníti a sémagráfot.

A mintaszám beállítása után a rendszer a sémára illeszkedő, random paraméterű eseménysorokat generál. A keletkező eseménysorlisták lesznek a neurális háló bemeneti adatsorai.

A mintarendszer Python-nyelven, Google Colab fejlesztési keretrendszerben készült.

A szekvencia előrejelzéséhez elkészítendő neurálishálózat-alternatívákat teszteltük öt változatban: két alapmodell és három új, saját változat. A két bázismodell a széles körben használt LSTM- és MLP-modellek. A javasolt módosítások az előtagsorozat hosszának a kiterjesztésére vonatkoznak, mely során nem szükséges az osztályozási hálózat bemeneti méretének kiterjesztése. A javasolt módszer egy egyedi előkészítő lépést alkalmaz a bemeneti vektor csökkentésére, mely lehet

- egyszerű szakszervezeti alapú csökkentés
- neurális hálózat alapú redukció

A modell a gráfot logikailag két szintre bontja: egy szinkronizációs, globális szint és egy ágens, aktor szint. A szinkronizálási, csatlakozási csomópont azt az esetet jelöli, amikor a több aktorszál folyamatai egymáshoz igazodnak. Ez lehet AND- és XOR-csomópont is. Ezek a vezérlő csomópontok automatikusan felfedezhetők, ha az eseménynapló tartalmaz egy objektum-/termékattribútumot is. Ez esetben a következő lépés megköveteli, hogy minden alkatrész rendelkezésre álljon, ami a bemeneten szükséges. A termékattribútum azonosítja a megmunkálás, cselekvés tárgyát. Ezzel a paraméter segítségével fedezhetjük fel a termék-/objektumszintű függőséget a különböző események között az eseménynaplóból.

Az aktoresemények mellett a betanítási folyamat kiterjesztett bemeneti eseménygráfja szinkronizációs vezérlő csomópontokat is tartalmaz, amelyek leírják a szomszédsági kapcsolatot az eseménysorozatok között. Feltételezzük, hogy minden A vezérlő csomópontnak van egy bemeneti eseménysorozata és egy kimeneti eseménykészlete is. A Petri-hálókhoz hasonlóan, a szinkronizációs vezérlő

csomópont csak akkor aktiválódik, amikor az összes bemeneti szekvencia befejeződött. Ha az átmenet elindul, minden kimeneti sorozatnál elindul a végrehajtás.

A vezérlő (AND- és XOR-) csomópontok bányászata a következőkön megfontolásokon alapul.

- Az egyes eseménycsomópontok feltárása, feldolgozása időbeli sorrendben, szekvenciában történik. A rendszer elsőként a szinkronizációs szekvenciát tárja fel.
- A vezérlő események szekvenciájánál a kimenő/bejövő objektumfüggőség által megszabott megszorításokra is tekintettel kell lenni.
- Eltérő aktorú események csatlakozásánál szinkronizációs csomópontra van szükség.
- A bejövő eseményszekvenciák alapján az első kimenő eredmény a szinkronizációs eseménysor feltárása.

A gráf sémaszintű modelljének a betanulása az alábbi lépéseken alapszik:

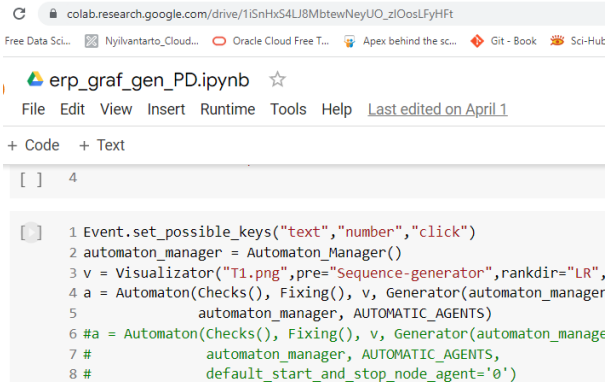
- az események láncának beolvasása
- a bemeneti/kimeneti termékfüggőség alapján a szinkronizációs pontok meghatározása
- a szinkronizációs (AND, XOR) eseményszekvenciák előállítása
- az eseményláncokból a (prefix szekvencia; következő esemény) tanítóhalmaz előállítása
- a tanítóhalmazok prefix részének redukciója a tanulóhálóknál
- szinkronizációs eseménylánc szekvencia modelljének meghatározása
- az aktorszintű szekvenciák, gráfrészletek kiemelése
- az eseményláncokból a (prefix szekvencia; következő esemény) tanítóhalmaz előállítása
- a tanítóhalmazok prefix részének redukciója a tanulóhálóknál
- aktorszintű eseménylánc-szekvencia modellek meghatározása

A gráf sémaszintű modell alapján történő predikció lépései:

- előzményrész inicializálása
- a szinkronizációs szekvenciamodell alapján predikció a soron következő szinkronizációs esemény meghatározására
- ezen lépések iterációja a végjelig
- a szinkronizációs eseménylánc előállítása
- az szinkronizációs események közötti aktorszintű eseményláncok típusainak meghatározása az eseményleíró adatbázis alapján
- az aktorszintű előzményrész inicializálása a háló már meglévő részei alapján

- az aktor-szekvenciamodell alapján predikció a soron következő aktoreseményre meghatározására
- ezen lépések iterációja a végjelig
- az aktoreseményláncok előállítása
- az elkészült láncok összefűzése egy közös eredménygráfba

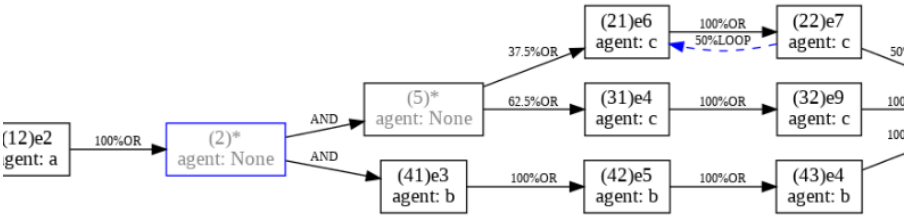
2.2. Eredményeket szemléltető képernyőképek



Séma generálási környezet

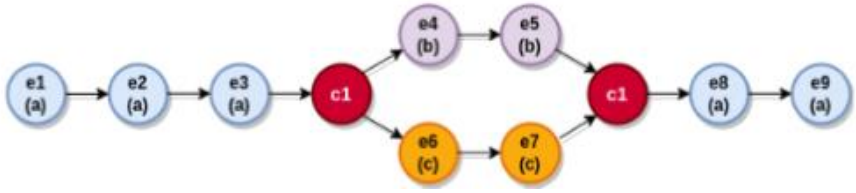
```
Event.set_possible_keys("text","number","click")
automaton_manager = Automaton_Manager()
v = Visualizator("T1.png",pre="Sequence-generator",rankdir="LR",debug =2)
a = Automaton(Checks(), Fixing(), v, Generator(automaton_manager),
              automaton_manager, AUTOMATIC_AGENTS)
#a = Automaton(Checks(), Fixing(), v, Generator(automaton_manager),
#              automaton_manager, AUTOMATIC_AGENTS,
#              default_start_and_stop_node_agent='0')
writer = FileWriter("file_Tuj.txt",pre="Sequence-generator")
#a.add_imaginary_node(new_node_id=1)
a.add_event(Event('e1',(1,2),{"text":"todo","click":True}), 0,
            new_node_id=11,
            agent_id='a')
a.add_event(Event('e2',(3,5),{"text":"todo","click":True}), 11,
            new_node_id=12, agent_id='a')
a.add_imaginary_node(12, new_node_id=2, new_node_type=AND)
a.add_imaginary_node(2, new_node_id=5, new_node_type=OR)
a.add_event(Event('e3',(1,4),{"text":"todo","click":True}), 2,
            new_node_id=41, agent_id='b')
a.add_event(Event('e5',(4,5),{"text":"todo","click":True}), 41,
            new_node_id=42, agent_id='b')
a.add_event(Event('e4',(2,3),{"text":"todo","click":True}), 42,
            new_node_id=43, agent_id='b')
a.add_event(Event('e6',(1,3),{"text":"todo","click":True}), 5,
            new_node_id=21,
            branch_possibilities=[0.3],
```

Sémagenerálás kódja



Előállított séma grafikus megjelenítése

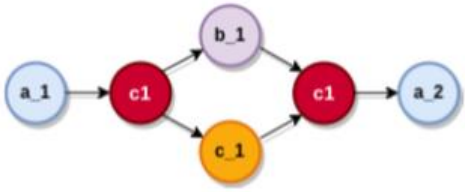
Kapcsolódó séma, amire a minták illeszkednek:



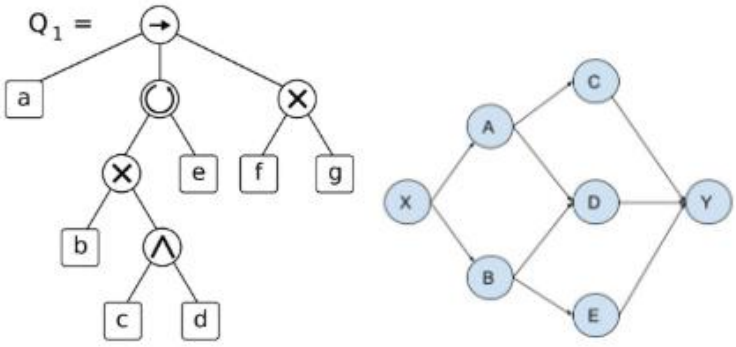
Generált mintaeseménysor:

```
EventT(1,1,'A',1)
EventT(2,2,'A',3)
EventT(3,3,'A',5)
EventT(4,4,'B',10)
EventT(5,5,'C',11)
EventT(6,6,'B',15)
EventT(7,7,'C',18)
EventT(8,8,'D',21)
EventT(9,9,'D',31)
```

Kapcsolódó szinkronizációs gráf:



2.3. Eredményeket szemléltető diagramok



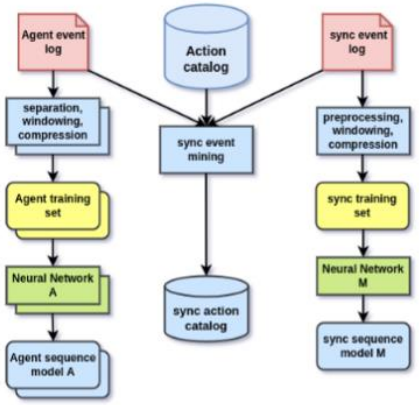
(a) Sample process tree [26] (b) Sample event graph

Table 1: Sample event log with artifact identification

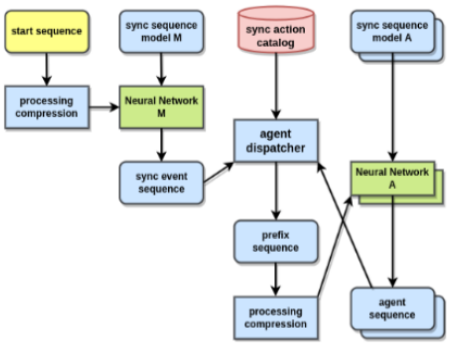
Trace	Actor	Action	Time	Input	Output
1	U1	A1	01.45	O1	O2
1	U1	A3	02.10	O2	O2
1	U2	A5	02.15	O2	O4,O5
1	U3	A8	02.18		O3
1	U3	A15	02.18	O3	O3
1	U4	A15	02.31	O2	O8
1	U4	A15	02.45	O8	O8

Az események szükséges leíró attribútumai

A tanuló NN-modul architektúrája



A predikciós NN-modul architektúrája



2.4. Kiértékelések eredményeinek bemutatása

Sequence generator rövid használati útmutató

A Sequence generator működtetéséhez elsőként a különböző osztályok példányosítását kell elvégezni. A fő kezelőosztály az *Automaton*, amiben a szerkezetet építhetjük, generálást hívhatunk meg. Az *Automaton* osztály konstruktora a többi osztály példányát várja függőségként.

*automaton.add_event(event,*parent_ids,new_node_type=,new_node_id=,branch_possibilities=[])*

Az első paraméter az esemény lesz, az ezt követő paraméterek vesszővel elválasztva a szülő csomópontok azonosítója. Az éppen létrehozott csomópont típusát, azonosítóját és a szülőktől az aktuális csomópontba vezető valószínűségek értékét a kulcsszavak kiírásával lehet csak megadni.

new_node_type esetében OR vagy AND típusú node-okról beszélhetünk (OR)
new_node_id esetében az adott node azonosítóját állítjuk be (automatikus)

branch_possibilities egy tömb, amely a sorrendben megadott szülőknél a valószínűség, hogy az éppen létrehozott csomópontba folytatódjon a szekvencia.

*automaton.add_imaginary_node(*parent_ids,new_node_type=,new_node_id=,branch_possibilities=[])*

Az add_eventhez hasonló, azonban az esemény helyére az IMAGINARY_NODE nevű konstanst állítja be. (add_eventtel megegyező)

automaton.add_event_between_nodes(parent_node_id,event,branch_possibilities,child_node_ids_list =, new_node_type=,new_node_id=)

ahol

parent_node_id: a szülő azonosítója
event: az adott node-ban tárolt esemény
branch_possibilities: tömb, amely a gyerekekhez kapcsolódás valószínűségét tárolja ugyanabban a sorrendben, mint ahogy a gyerekeket megadtuk
child_node_ids_list: a gyerekek azonosítója egy tömbben [3,10,32] (Az összes gyerek)
new_node_type: OR vagy AND típusú legyen-e a node (OR)
new_node_id: az új köztes node azonosítója (automatikus)

Elágazási valószínűség beállítása

automaton.reset_branch_possibilities_in_order(node_id,branch_possibilities)

ahol

a node_id azonosítójú node gyermekeinek valószínűségét módosítja;
node_id: node azonosító szám
branch_possibilities: tömb, amiben a legkisebb gyermekazonosítótól a legnagyobbig kell megadni a valószínűségeket. A tömbben megadott valószínűségek összege 1 kell hogy legyen, például: automaton.reset_branch_possibilities_in_order(2,[0.2,0.8])

A példa esetében a valószínűségek a következőképpen fognak kinézni.

Új él felvitele:

automaton.add_node_connection(start_node_id,end_node_id,branch_possibility,is_loop=)

A parancs két csomópontot köt össze az azonosítói alapján a megadott valószínűséggel. Az is_loop= adattag alapértéke false, szükséges, mert ha a kapcsolat referenciakört eredményez, végtelen ciklusba jutunk, ha ezt az adattagot igazra állítjuk, akkor a rendszer a végigiterálások alatt nem fogja vizsgálni ezt az ágat.

automaton.add_loopback(start_node_id,end_node_id,branch_possibility):

Az előző metódust hívja meg úgy, hogy itt egyértelműen loop típusú lesz a kapcsolat.

automaton.end_branch(last_node_id,end_possibility):

A last_node_id azonosítójú csomópontot a záró csomóponthoz köti egy *end_possibility* valószínűséggel.

automaton.visualize()

A gráf vizualizációját menti el a Visualize("file.png",pre=) osztály példányosításánál megadott helyre (alapesetben a generated_data/graphs/file.png).

automaton.finalize()

Lefuttatja az ellenőrzéseket és automatikusan kijavítja a következő hibákat. Ha egy node-nak egy gyereke van és a valószínűsége nem 1; Ha egy node-nak nincs gyereke, mégsincs hozzákötve a záró node-hoz.

automaton.generate(number_of_sequences):

A szerkezeten szimulációt futtat number_of_sequences darabszor, és lementi az utat egy tömbökből álló kódolt formára.

automaton.non_trivial_auto_fixing(node_id_list):

node_id_list: lista node id-kel, ahol a gyerekek valószínűségét magától beállítja (arányszámokként veszi a valószínűségeket).

A Sequence_Manager osztály főbb funkciói

create_agent_lists(generated)

Az automaton.generate() metódus által generált adatokat ágensek által végrehajtandó csomópontok tömbjére konvertálja.

create_sequence_objects(sequence_list)

A *create_agent_lists()* által kiadott listákból Sequencia listát csinál.

FileWriter writeSequence(seqence_list)

A create_sequence_objects által visszatért Sequencia listát a FileWriter(path_and_file,pre=) konstruktorában megadott path_and_file fileba írja ki.

Visualizator osztály

show_tree_colab()

Megjeleníti a gráf fileját a colab felületén.

Hibaüzenetek típusai

PossibilitySumException: kiírja, hogy mely node-ok esetében helytelen a valószínűségek összege.

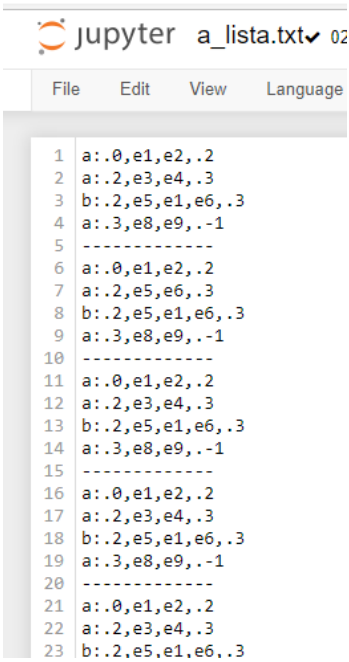
FailedStructureCheckException: általános strukturális hiba, a hibaüzenetben a hiba részletei is le vannak írva.

StructureNotClosedException: Amikor nincs egy node sem a záró node-hoz kötve. (Ha az utolsó node loop is.)

LoopNotAvailableException: Egy AND típusú node-ra szerettük volna a loop kezdetet rakni, ami így végtelen ciklushoz vezetne.

InvalidNodeIdException: A megadott felhasználó által beállított node_id már foglalt.

3. Technológiai folyamatok bemutatása



Generált eredménylista, a sémára illeszkedő random eseménysorokkal

Table 2: Accuracy comparison of the sequence prediction networks

Dataset	LSTM	MLP	NH-MLP	BE-MLP	BE_LSTM
$pdc_2016_1.xes$	63.5	63.4	63.4	63.9	64.1
$load_random$	58.2	57.5	56.6	58.7	58.0
$pdc_2016_9.xes$	82	82.5	81.2	81.8	82.6
$pdc_2017_5.xes$	62.4	62.8	63.7	64.8	64.2
$pdc_2019_2.xes$	64.6	65.2	63.9	66.27	65.6

Az LSTM-/MLP-teljesítmény-összehasonlító mérések eredményei (accuracy értékek). A mérések jól mutatják az MLP-változat (BELSTM) előnyeit:

- kisebb komplexitás
- gyorsabb végrehajtás
- ez egyik legjobb hatékonyság

A következő kódrészletek a három vizsgált hálótípus megvalósítását leíró kódot mutatják be.

```
                                # alap LSTM-háló
class lstm_A:

    def __init__(self,Lin, Lmid, Lout):
        self.model = Sequential()
        self.model.add(
            LSTM(Lmid,
                activation='relu',
                input_shape=(1, Lin))
        )
        self.model.add(Dense(Lout, activation="softmax"))
        self.model.compile(optimizer='adam', loss='categorical_crossentropy',metrics=['accuracy'])
        #self.model.summary()

    def train (self, Xtr,Ytr,ep):

        self.model.fit(Xtr,Ytr,epochs=ep, verbose=0)
        self.model.fit(Xtr,Ytr,epochs=1, verbose=1)


    def predict (self, Xte, Yte):

        Yge = self.model.predict (Xte)

        db = 0
        for i in  range(Yge.shape[0]):
            yg = list(Yge[i,:])
            yt = list(Yte[i,:])
            if yg.index(max(yg))  == yt.index(max(yt)):
                db += 1
        print ("accuracy:",db/Yge.shape[0] )


                                # alap MLP háló
```

```
class mlp_A:

    def __init__(self,Lin, Lmid, Lout):
        self.model = Sequential()
        self.model.add(
            Dense(Lmid,
                activation='relu',
                input_shape=(Lin,))
        )
        self.model.add(Dense(Lout, activation="softmax"))
        self.model.compile(optimizer='adam', loss='categorical_crossentropy',metrics=['accuracy'])
        #self.model.summary()

    def train (self, Xtr,Ytr,ep):

        self.model.fit(Xtr,Ytr,epochs=ep, verbose=0)
        self.model.fit(Xtr,Ytr,epochs=1, verbose=1)


    def predict (self, Xte, Yte):

        Yge = self.model.predict (Xte)

        db = 0
        for i in  range(Yge.shape[0]):
            yg = list(Yge[i,:])
            yt = list(Yte[i,:])
            if yg.index(max(yg))  == yt.index(max(yt)):
                db += 1
        print ("accuracy:",db/Yge.shape[0] )

                                    # saját nested MLP-háló
class nhmlp_A:

    def __init__ (self, X, Y, lra=0.01, P1=3,P2=6):

        L = len(X) # L : szintek száma
        inp_A = []  # bementi rétegek
        dense_A1 = []  # első rejtett rétegek
        dense_A2 = [] # második rejtett rétegek
        out_A = []  # kimeneti rétegek
```

```
    inps = []   # az összesítő háló bemenete

    for l in range(L):              # ciklus a rétegekre

      d1 = X[l].shape[1]            # bemeneti réteg
      d2 = Y.shape[1]
      inp_A.append(tf.keras.Input(shape=(d1,)))
                                    # első köztes réteg
      dense_A1.append( layers.Dense(P1*d2, activation="relu"))
      x1 = dense_A1[l](inp_A[l])
                                    # második köztes réteg
      dense_A2.append(layers.Dense(d2, activation="relu"))
                                    # kimenetek
      out_A.append( dense_A2[l](x1))

                                    # kimenetek összefűzése
    outs = []
    for l in range(L):
      outs.append(out_A[l])
                                    # bemenet az összesítő hálórészhez
    inp =  layers.concatenate(outs)
                                    # összesítő háló első köztes rétege
    d3 = Y.shape[1]
    dense_1 = layers.Dense(P2*d3, activation="relu")
                                    # összesítő háló második köztes rétege
    dense_2 = layers.Dense(d3, activation="softmax")
    x = dense_1(inp)
    out =  dense_2(x)               # kimenet
                                    # eredő háló összeállítása
    self.model = tf.keras.Model(inputs=inp_A, outputs=out)
    #self.model.summary()
    aopt = tf.keras.optimizers.Adam(lr=lra)
    self.model.compile(loss=tf.keras.losses.CategoricalCrossentropy(),metrics=["accuracy"], opti-
mizer=aopt)

  def train (self, Xtr,Ytr,ep):

    self.model.fit(Xtr,Ytr,epochs=ep, verbose=1)
    self.model.fit(Xtr,Ytr,epochs=1, verbose=1)
```

```
def predict (self, Xte, Yte):

    Yge = self.model.predict (Xte)

    db = 0
    for i in  range(Yge.shape[0]):
        yg = list(Yge[i,:])
        yt = list(Yte[i,:])
        if yg.index(max(yg))  == yt.index(max(yt)):
            db += 1
    print ("accuracy:",db/Yge.shape[0] )
```

4. Összegzés

- Azonosítottama kezelendő gráfmodell szükséges vezérlőelemeit.
- A kapott eredmények alapján olyan neurális háló modell kell, ami támogatja az AND, XOR vezérlési elemeket.
- Elkészült a eseménygráfséma logikai modellje.
- Elkészült a sémaparaméterezés modellje.
- Elkészült a sémageneráló keretrendszer.
- Elkészült a sémára illő random eseményláncok generálását végző rendszer.
- Megterveztük és implementáltam az LSTM- és MLP-alapú hálókat.
- A gráfpredikciós motorhoz kiválasztásra került az MLP-alapú hálótípus.
- Elkészült az a háló séma feltárási modell, amely alkalmas az AND, XOR vezérlési elemek meghatározására is.
- Kétszintű feldolgozási modell kidolgozása a komplex eseménygráf feltárására.
- Algoritmus kidolgozása a szinkronizációs események feltárására.
- Algoritmus kidolgozása a prefix rész tömörítésére.
- Neurális háló alapú gráfpredikciós motor kidolgozása, amely alkalmas az AND, XOR vezérlési elemek meghatározására is.

SZÖVEGOSZTÁLYOZÓ MÓDSZEREK VIZSGÁLATA ÜGYFÉLSZOLGÁLATI KÉRÉSEK ELŐFELDOLGOZÁSÁHOZ

CSÉPÁNYI-FÜRJES LÁSZLÓ

A gyakori üzleti folyamatok feltárásához szükséges az eseménynaplók elemzésén túl más ügyfélkapcsolati csatornák tanulmányozása is. Ezen csatornák egyik legfontosabb eleme az e-mail. Mivel az e-mailek úgynevezett strukturálatlan szöveges formában léteznek, ezért kézenfekvő, hogy NLP-/NLU-technikákat vessünk be az üzleti folyamatok azonosítására, illetve feltárására. A kutatási modul célja, hogy az NLP-/NLU-szöveg tartalomosztályozási módszereit áttekintse és javaslatot tegyen egy szabad szöveges előfeldolgozó rendszer kidolgozására. Az ügyfelektől származó megkeresések előfeldolgozásával az ügyintéző munkája hatékonyabbá tehető, több idő szánható magára az ügyféllel végzett kommunikációra, illetve magára a feladat elvégzésére.

1. A kutatás célja és lépései

Az kutatás egy olyan koncepció kidolgozására irányult, mely alap lehet egy magyar nyelvű szövegosztályozó rendszer felépítéséhez. Ez a tervben megfogalmazott minimumcél eléréséhez volt szükséges. Egy konkrét nyelvi modell, a RoBERTa (Robustly optimized Bidirectional Encoder Representations from Transformers) alkalmazhatóságát jártuk körbe, mivel a szakirodalom tanulmányozása után ez tűnt a kézenfekvő választásnak, tekintve a rendelkezésünkre álló nyelvi korpuszok korlátozott terjedelmét, valamint azt a számítási teljesítményt, amit a kutatás során felhasználhatunk.

Technikai szempontból a cél egy olyan rendszer kidolgozása, mely képes magyar nyelvű strukturálatlan inputból feltárni az ügyfél lehetséges szándékait, ezen szándékokat pedig megfelelő folyamatazonosítókhoz képes rendelni.

- Minimumcél, hogy az inputot egy konkrét szándékfeltárás (intent detection) segítségével egy folyamat-azonosítóhoz kapcsoljuk.
- Maximumcél, hogy az inputból többszörös szándékot tárjunk fel (multi intent detection), valamint az egyes szándékokhoz kapcsolódó információkat (nevek, azonosítók, dátumok stb.) is kinyerjük (slot labeling).
- Célként megfogalmazódott, hogy doménspecifikus annotált tanítóminta hiányára is dolgozzunk ki egy eljárást. Tegyünk javaslatot, hogyan tudja az üzleti partner ezen tanítómintát saját hatáskörében, saját erőforrásának bevonásával elkészíteni.

1.1. Alapmodell létrehozása

A következő konkrét vizsgálatokat végeztük el, melyek során arra kerestük a választ, hogy alkalmas-e, és ha igen milyen formában az egyik legjelentősebb magyar nyelvű annotált korpusz, a Szeged Dependency Treebank (SZDT) arra, hogy segítségével szándékdetektálásra alkalmas nyelvi modellt hozzunk létre.

- Az elemzés során egy 10 000 mondatos alkorpuszt hoztunk létre (SZDT–10000) és az annotált formátumot annotálatlan szöveges formátumra alakítottuk, valamint elkészítettünk egy társállományt, mely a mondatok függőségi nyelvtanból ismert ún. gyökér (ROOT) fogalmát tartalmazza. Ezen két állomány segítségével szándékmegállapítás-kísérletet lehet végezni, mégpedig élve azzal a feltételezéssel, hogy a mondat a gyökér szava által definiált szándékot indukálja.
- Egy még szűkebb, 7000 mondatos alkorpuszon hoztunk létre egy előtanított (pre-trained) RoBERTa modellt, melynek az SZDTROBERTA–7000 munkanevet adtuk.
- Elvégeztük az elkészült modell szövegosztályozásra történő finomhangolását és kiértékelését.
- Ugyanezen kísérletet az angol nyelvű, NLU-BENCHMARK tanítómintán is elvégeztük, a ROBERTA-BASE előtanított modell felhasználásával.
- A Nyelvtudományi Intézet HIL_ROBERTA modelljét is bevontuk a vizsgálatokba arra keresvén a választ, hogy a mi előtanított modellünk versenyképes megoldást nyújt-e.

A kísérletekhez a PyTorch-könyvtárat használtuk. Egy Nvidia GTX 1050Ti GPU-val rendelkező Linux-rendszeren futtattuk az elemzéseket.

A kísérletek során használt modellek leírásai, valamint cikkek referenciái az *intent_prediction/README.md* fájlban kerültek összegyűjtésre.

A kis méretű SZDTROBERTA–7000 modell tesztelésének tanulságait a következő fázisban arra használtuk fel, hogy egy továbbfejlesztett, nagy méretű előtanított modellt állítsunk elő. A kísérletek folytatásához az intézet úgynevezett *dataminer* szerverét jelöltük ki, melyen kialakítottuk a kísérleti környezetet. Ez az erőforrás már rendelkezik GPU-val, ami lehetővé tette számunkra, hogy betanítsunk egy nagyobb méretű alapmodellt, melynek az SZDTROBERTA munkanevet adtuk.

1.2. Az NLU_BENCHMARK magyar nyelvű megfelelőjének elkészítése

További kísérletek elvégzése céljából szükségünk volt egy magyar nyelű osztályozott szövegre, mely tanításra, illetve tesztelésre egyaránt felhasználható. Elhatároztuk, hogy az angol nyelvű 7 osztályos NLU_BENCHMARK tanítómintát automatikusan magyar nyelvre konvertáljuk, és ezzel fejlesztjük tovább a kísérleti rendszerünket, mely ez idáig csupán a mondat ROOT szavának predikciójára volt képes. A konvertált modellnek NLU_BENCHMARK_HU_RAW munkanevet adtuk.

Automatikus fordításhoz a Google *translate* Python API-t, és magát a translate szolgáltatást is felhasználtuk, de az eredmény nem volt kielégítő, mivel karakterkódolási és nyelvhelyességi hibák egyaránt keletkeztek. A keletkezett tanítóminta karakterkódolási problémáit kijavítottuk. Az előállt fájlt ezután manuálisan sorról sorra átnéztük, és az ott található szöveget kijavítottuk. Legjellemzőbb problémaként említhető a fordító hibájából keletkezett sok magyartalan kifejezés. A végleges tanítómintának az NLU_BENCHMARK_HU munkanevet adtuk.

1.3. A saját kísérleti alapmodelljeink és a HIL_ROBERTA alapmodell összehasonlítása

Az SZDTROBERTA–7000 kísérleti alapmodellt és a HIL_ROBERTA alapmodellt behatóbban össze kellett hasonlítanunk. Ennek két célja volt:

1. A HIL_ROBERTA licenc birtokosa a Nyelvtudományi Intézet, ami behatárolja, hogy mire használhatjuk a modellt. E helyett tehát érdemes egy saját alapmodellt használni.
2. A kísérleteink szempontjából az SZDTROBERTA–7000 egy doménspecifikus alapmodellnek tekinthető. Bár a HIL_ROBERTA valószínűleg jobb általános célra, de doménspecifikus feladatokra a saját alapmodell is megfelelő lehet.

A magyar NLU_BENCHMARK_HU_RAW tanítóminta segítségével 4 finomhangolt modell tanítását végeztük el, és az elkészült modelleket manuálisan is kiértékeltük.

A javított magyar NLU_BENCHMARK_HU tanítóminta segítségével a tanítást 3, illetve 6 epochon keresztül újra elvégeztük, először az új SZDTROBERTA modell, majd a HIL_ROBERTA modell finomhangolásával.

1.4. Kapcsolt szándékfeltárás és adatkinyerés

A magyar nyelvű saját alapmodell elkészülte után a fókuszunkat a többszörös szándék-prognosztizáció, illetve a szándékhoz kapcsolódó adatok kinyerésére fordítottuk. Olyan megoldásokat kerestünk, ahol ezen feladatok kapcsolt módon kerülnek megoldásra.

2. Kutatási eredmények összesítése

2.1. Elvégzett kísérletek bemutatása

2.1.1. Alapmodell létrehozása

A magyar nyelvű tanítóminta kiválasztása, előkészítése

A magyar nyelvű kérések feldolgozásához magyar nyelvi modellre, illetve magyar tanítómintára van szükségünk. Az egyik legjelentősebb magyar nyelvű annotált korpuszt, a Szeged Dependency Treebank (SZDT) felhasználását jelöltük ki a kezdeti vizsgálat tárgyaként. Az SZDT beszerzése megtörtént, ebből egy 10 000 mondatos alkorpuszt készítettünk elő a szabad szöveges feldolgozásra. Ezen folyamat során az annotált formátumot annotálatlan szöveges formára alakítottuk, valamint elkészítettünk egy társállományt, mely a mondatok függőségi nyelvtanból ismert, ún. ROOT fogalmát tartalmazza. Ezen két állomány szükséges a szándék-prognosztizációs kísérlet elvégzéséhez. A kísérlet alapja az a feltételezés, hogy a mondat a gyökér szava által definiált szándékot indukálja. Ez a gyökércímke helyettesíti a mondat szándékleíró kódját. Mivel ekkor nem állt rendelkezésre konkrét magyar nyelvű osztályozott tanítóminta-korpusz, ezért a leírt módszerrel kezdtük el a vizsgálatokat.

	utterance	label
9995	Pár perc múlva apa talált egy csavarkulcsot.	talált
9996	Jól összekente vele magát.	összekente
9997	Úgy gondolta, hogy a kilátóhelynél elrejti, vi...	gondolta
9998	Csodálatos volt a kilátás, a hegyeken látszott...	volt
9999	Anya talált egy szarvasbogarat, ezzel ő vezetett.	talált

Részlet az elkészült tanítómintából

A 10 000 mondat ezzel a módszerrel 2924 osztályba volt sorolható. Az előtanított modell a SZDTROBERTA–7000 volt, melyet egy korábbi NLP-kísérletünk során hoztunk létre.

Angol nyelvű tanítóminta kiválasztása, előkészítése

Az úgynevezett NLU-BENCHMARK halmaz alkalmas arra, hogy angol nyelvű szöveg-osztályozó, illetve feladatmegoldó algoritmusokat értékeljünk ki. Összesen 7 osztályt tartalmaz, melyek egyben szándékleíró címkéknek is tekinthetőek.

Utterance	Intent
I would like to hear The Worst Is Yet To Come	*PlayMusic*
Give A History of the Mind a 2 out of 6 points.	*RateBook*

Példák az NLU-BENCHMARK-ból

Minden osztályban 300 mondattöredék (utterance) található:
Class: AddToPlaylist, # utterances: 300
Class: BookRestaurant, # utterances: 300
Class: GetWeather, # utterances: 300
Class: PlayMusic, # utterances: 300
Class: RateBook, # utterances: 300
Class: SearchCreativeWork, # utterances: 300
Class: SearchScreeningEvent, # utterances: 300

Ezen szándékleíró címkéket azonosítókká konvertáltuk:
label to index: {'AddToPlaylist': 0, 'BookRestaurant': 1, 'GetWeather': 2, 'PlayMusic': 3, 'RateBook': 4, 'SearchCreativeWork': 5, 'SearchScreeningEvent': 6}

Előtanított modellként az angol nyelvű ROBERTA-BASE-t használtuk:
config = RobertaConfig.from_pretrained('roberta-base')

Kísérletek a magyar és az angol nyelvű tanítóminták segítségével

NLU-BENCHMARK tanítóminta jellemzői:
FULL Dataset: 2100
TRAIN Dataset: 1680
TEST Dataset: 420
Intent: 7

SZDT–10000 tanítóminta jellemzői:
FULL Dataset: 10 000
TRAIN Dataset: 8000
TEST Dataset: 2000
Intent: 2924

A tanítás az angol nyelvű minta segítségével 3 epochon keresztül zajlott, mely eredménye egy kevéssé rugalmas túltanított modell lett:
Iteration: 1600. Loss: 0.03266046196222305. Accuracy: 100,0%

A magyar nyelvű minta a SZDTROBERTA–7000 használatával, karakter alapú tokenizálással nem hozott értékelhető eredményt:
Iteration: 4600. Loss:7.135164566040039. Accuracy: 10,1%

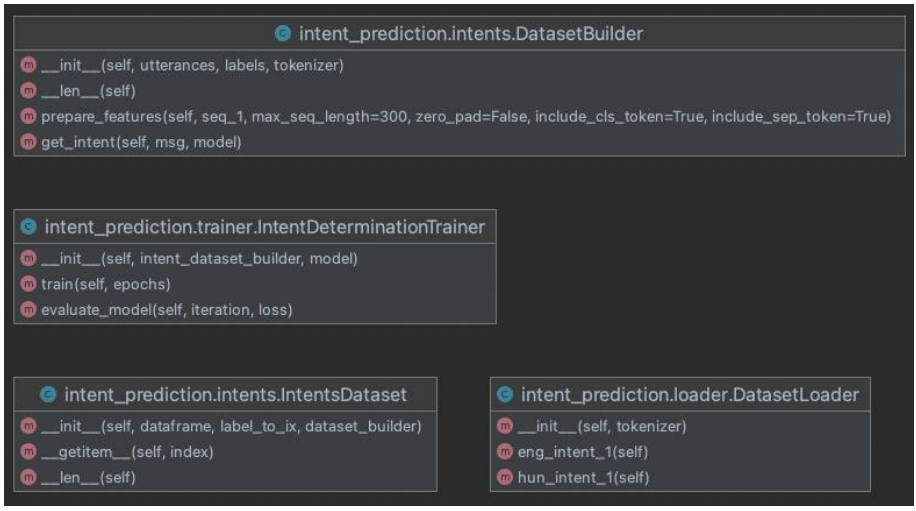
A magyar modell újratanítása teljes szavas tokenizálással

A magyar modellt ismételten betanítottuk teljes szavas tokenizálással. Ennek eredménye lett a **model_2,** mely már értékelhető pontossággal bírt (Accuracy: 36,15%).

Ugyanezen tanítómintával a HIL_ROBERTA modellt is finomhangoltuk, így össze tudjuk mérni az általános alapmodell (HIL_ROBERTA), valamint a doménspecifikus alapmodell (SZDTROBERTA–7000) teljesítményét. A HIL_ROBERTA alapmodell finomhangolását háromféleképpen végezetük el, folyamatosan emelve az epochok számát: 3-ra, 6-ra, majd 12-re.

Python-osztályok

A kísérlet során kifejlesztésre került programrészeket objektumorientált struktúrába szerveztük, mely megkönnyíti a további kísérletek elvégzését, illetve az alrendszer beépítését egy nagyobb struktúrába. A kétnyelvű kísérlet jelenleg a következő osztályokra épül:



Python-osztályok

Az SZDTROBERTA modell elkészítése a teljes SZDT-korpusz felhasználásával

A *dataminer* szerver használata saját, nagy méretű ROBERTA modell elkészítésére: A tanítás utasításait a *roberta.ipynb* notebookfájl tartalmazza. A megfelelő modulok telepítése után egy saját *ByteLevelBPETokenizer* tanítását végeztük el. Ezúttal a teljes SZDT-korpuszt felhasználtuk. Néhány manuális teszteléssel ellenőriztük le a tokenizer működését.

A *RobertaForMaskedLM* Python-modellt előtanítottuk, mely a megadott konfigurációban 83.5 millió paramétert tartalmaz. A definiált transformer tokenizer segítségével létrehoztunk egy data collatort. A collator feladata, hogy a tanítóminta adatait a *Trainer* számára megfelelően kötegelt formátumúra hozza. Végül megadtuk a *Trainer* megfelelő argumentumait és elindítottuk a műveletet.

Mivel GPU-hibák jelentkeztek, elemeztük a beállításokat és kiderítettük, hogy jelen formában a köteg méretét egészen 4-re le kell csökkenteni, hogy a folyamat sikeres legyen:

```
per_device_train_batch_size=4
```

Tanulságként levonható, hogy a folyamatot érdemes először CPU-n kipróbálni, és csak miután hibamentesen elindul a tanítás, akkor aktiválni a GPU-t. Ennek oka, hogy a CPU-val kapcsolatos műveletek sokkal beszédesebb, konkrétabb hibaüzeneteket adnak, amivel jelentősen megkönnyítik az elemzést, hibajavítást.

A kísérletsor célját elértük, egy saját nagy méretű RoBERTa modell létrejött, melynek a SZDTROBERTA munkanevet adtuk.

2.1.2. Az NLU_BENCHMARK magyar nyelvű megfelelőjének elkészítése

Angol nyelvű 7 osztályos tanítóminta konverziója

Feltételeztük, hogy a Google Translate online fordítóalkalmazás (https://translate.google.com) megfelelő választás lesz a tanítóminta konvertálására. Mivel az angol nyelvű minta nem nyers szövegként áll rendelkezésre, ezért az volt a terv, hogy egy Python könyvtár segítségével hajtjuk végre a fordítást. Választásunk a *googletrans* könyvtárra esett:

```
from googletrans import Translator
```

Azonban úgy tűnik a Google Translate backend nem támogatja a nagy tömegű elérést, ezért néhány sikeres request-response után a fordítási folyamat hibával megszakadt.

A tömeges mondatfordítás lehetőségét így elvetettük és helyette exportáltuk mind a 2100 mondattöredéket:

intent_dataset_builder.dataset['utterance'])

Mivel ekkora mennyiségű szöveget nem támogat a Google Translate fájlként sem, ezért 500 soros fájlokra daraboltuk azt, majd az elkészült eredményfájlokat összefűztük. Az elkészült tanítómintát ezután tisztítani és véglegesíteni kellett, hogy használható legyen modell-finomhangolásra. Tömörítés után az állományok bekerültek a git repositoryba a következő néven:

translated_intent.zip
translated_intent_labels.zip

Az adatbetöltő modult frissítettük ezekkel az új állománynevekkel és elkészítettük az új tanító jupyter notebookot:

train_hun_intent_model_2.ipynb

Az elkészült tanítómintának az NLU_BENCHMARK_HU_RAW munkanevet adtuk.

2.1.3. A saját kísérleti alapmodelljeink és a HIL_ROBERTA alapmodell összehasonlítása

Az SZDTROBERTA–7000 és a HIL_ROBERTA alapmodelleket több tanítóminta finomhangolása segítségével is összehasonlítottuk. Az SZDT–10000 minta volt az első kísérletünk tárgya. Többféle tokenizációs eljárást, illetve többféle epoch beállítást is teszteltünk, a cél egyetlen szándék, a mondat ROOT kifejezésének meghatározása volt.

A két alapmodell összehasonlítását az automatikusan magyar nyelvre fordított NLU_BENCHMARK_HU_RAW tanítóminta felhasználásával folytattuk. Itt már képesek voltunk a definiált 7 szándék detektálási sikerességének mérésére, illetve összehasonlítására. Továbbá néhány saját mondatot is megadtunk, melyek segítségével manuális összehasonlítást is végeztünk.

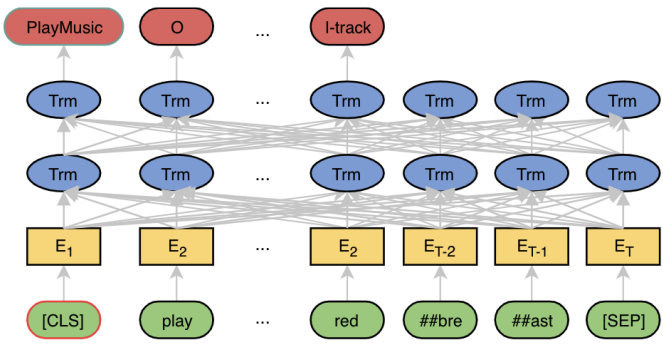
Az NLU_BENCHMARK_HU_RAW tanítóminta manuális javításával előállt NLU_BENCHMARK_HU tanítóminta segítségével, valamint a nagy méretű SZDTROBERTA alapmodellünk felhasználásával 4 modell finomhangolását végeztük el, és az elkészült modelleket manuálisan is kiértékeltük, valamint összevetettük a HIL_ROBERTA alapmodell finomhangolásából származó eredményekkel. Konklúzióként megállapítottuk, hogy a saját alapmodellünk fölveszi a versenyt a Nyelvtudományi Intézet modelljével, így a továbbiakban a saját alapmodellünket fogjuk a kísérletekre használni.

2.1.4. Kapcsolt szándékfeltárás és adatkinyerés

Az irodalom alapján Gangadharaiah et al. a szándékdetektálási feladatot mint többszörös címkézés (multi-label classification) oldja meg. Xu et al. (2013) munkájára építenek, akik a feladathoz log-linear módszert használnak. Kiegészítésképpen Gangadharaiah et al. slot labeling műveletet is végeznek. Példaként tekintsük a következőket: ugyanazon domén két különböző szándékot is magában foglal: „BookCab" és „BookHotel". A „BookCab" szándékhoz a következő három label kapcsolható: „City", „Time" és „Loc". A „BookHotel" szándékhoz a „City", „CheckinDate" és a „Duration". A következő szövegben: „book a cab from the airport in Seattle and find me a hotel to stay" a felhasználó hotelt és taxit is foglalni kíván, vagyis mindkét szándék aktív. A megjelenő „City" label: „Seattle" mindkét szándékhoz kapcsolódik. A feladatok megoldásához BiLSTM neurális hálózatot használnak encoder-decoder összeállításban. Szóbeágyazásként egy külön réteget használnak előtanított beágyazások helyett.

Chen et al. előtanított BERT-modell felhasználásával oldja meg a szándékprognosztizációs feladatot. Javaslatuk szerint egy speciális klasszifikációs tokent ([CLS]) illesztenek a szószekvencia elejére, valamint egy elválasztó ([SEP]) tokent a szekvencia végére. A [CLS] token hidden state-jét használva prognosztizálható a szándék. A többi token hidden state-jének felhasználásával állapítható meg a slot label. Mivel WordPiece tokenizálót használnak, ezért a tokenek első sub-Tokenjének hidden state-jét használják a teljes tokenek helyett. Javasolt megoldásukkal kimagasló eredményeket produkálnak.

A transformer modell bemeneti oldalán nem csupán a mondatok tokenizált formája jelenik meg, hozzáfűzésre kerül egy speciális token a mondat elejére [CLS] valamint a végére [SEP].



A Transformer-modell tokenkezelése

Számunkra a [CLS] token a legizgalmasabb. Az ábra bemutatja, hogy a transformer modell outputján a [CLS] token a szándékleíró label értékét veszi fel. Igaz, hogy a [CLS] token értéke a bemeneti oldalon mindig ugyanaz, azonban a transformer modell környezetfüggő (contextualized), így a kimenet a szövegkörnyezettől függően más és más lehet. Finomhangolás során tehát ezzel a módszerrel vagyunk képesek a mondat klasszifikációjára.

Mint látszik, a többi mondatalkotó token (pl.: szó, írásjel) adatleíró címkévé transzformálódik. Ez a két feladat (intent detection & slot filling) tehát kapcsolt módon oldható meg a transformer modell segítségével.

Az elemzések következő fázisában elkezdtük a CLS-token használatának beépítését a kísérleti rendszerünkbe. A *DatasetBuilder* osztály, mely előkészíti a nyers szövegből az inputvektort, tartalmaz egy *prepare_features()* függvényt, mely paramétertől függően hozzáragasztja az inputvektorhoz a CLS- és SEP-tokenek reprezentációját. Azért, hogy ezen függvény működése érthetőbb legyen, implementáltunk hozzá egy egységtesztet: *test_data_loader()*.

Mivel a kísérleti rendszerünk modellstruktúrája egyszeres szándékfeltárásra készült, ezért ebben a formájában nem alkalmas arra, hogy kapcsolt többszörös szándékfeltárást és szekvenciális címkézést végezzünk vele. Jó alapot biztosít azonban a továbbfejlesztésre. A GitHubon elérhető JointBERT algoritmus forráskódelemzése megmutatta, hogy a kapcsolt végrehajtás megoldható BERT, DistilBERT és alBERT modellek esetén, ezért jó esély van arra, hogy RoBERTa modellnél is alkalmazható. A kapcsolt modell lényegét tekintve a kimeneten mind az *intent*, mind a *slot* megjelenik, melyhez párosul az előző réteg hidden_state-je is. Ezt a megoldást kell beépítenünk a RoBERTa modellbe is.

```
outputs = ((intent_logits, slot_logits),) + outputs[2:]  # add hidden states and attention if they are here

outputs = (total_loss,) + outputs

return outputs  # (loss), logits, (hidden_states), (attentions) # Logits is a tuple of intent and slot logits
```

A tanítóminta módosítására is szükségünk van, mivel a mondattöredékek, illetve szándékkategóriák meghatározása mellett a szekvenciális címkéket is felhasználunk. A következő példák az ún. ATIS (Airline Travel Information System) adathalmazból származnak. Jól látható, hogy a mondattöredék, a hozzá tartozó kategória, valamint a szekvenciális címkék listája 3 fájlban kerül tárolásra. Összekötésük a sorszám alapján történik.

```
18  i'm looking for a flight from oakland to denver with a stopover in dallas fort worth
19  what's restriction ap68
20  what types of ground transportation are available in philadelphia
21  what does the abbreviation co mean
22  a first class flight on american to san francisco on the coming tuesday
```

```
18  atis_flight
19  atis_restriction
20  atis_ground_service
21  atis_abbreviation
22  atis_flight
```

```
18  O O O O O O B-fromloc.city_name O B-toloc.city_name O O O O B-stoploc.city_name I-stoplo
19  O O B-restriction_code
20  O O O O O O O O B-city_name
21  O O O O B-airline_code O
22  O B-class_type I-class_type O O B-airline_name O B-toloc.city_name I-toloc.city_name O O
```

Ahhoz, hogy a szükséges tanítómintát elkészíthessük, szükségünk van egy annotációkészítő alkalmazásra. A finomhangolás során a transzformer modell bemeneti oldalán szakterületspecifikus mondatok jelennek meg (text), melyhez egy annotációszekvenciát párosítunk (labels). A szöveg ellátása annotációval egy monoton feladat, melyre számos megoldás létezik napjainkban. Ezen megoldások áttekintését elvégeztük. Az elemzés során megpróbáltuk azokat a kategóriákat áttekinteni, melyek számunkra leginkább relevánsak voltak.

A fent említett kapcsolt megoldás még nem teljesen fedi le a kutatási tervben ismertetett kívánalmakat, ezért tanulmányoztuk, hogyan tudnánk a többszörös kategorizálást, vagyis a többszörös szándékprognosztizálást megvalósítani. Ebben az esetben a kategória egy vektorként definiálható, mely azokon a pozíciókon veszi fel az 1 értéket, ahol a pozíciónak megfelelő kategória a mondatra érvényes, a többi pozíción 0. Jó példa erre, az angol nyelvű ToxicComments adathalmaz.

```
train_df = pd.read_csv('data/train.csv')
train_df.head()
```

Out[2]:

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
	32d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
	d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
	7ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
	1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
	54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0

2.2. Kiértékelések eredményeinek bemutatása

2.2.1. SZDT–10000 tanítómintán végzett kísérletek

Az SZDTROBERTA–7000 és a HIL_ROBERTA alapmodellek segítségével végzett egyszeres szándékmeghatárolási kísérletek eredménye azt mutatta, hogy a Nyelvtudományi Intézet modellje jelentősen pontosabb, mint a saját kis méretű modellünk.

Model ID	Tokenizer and model	Tokenize	Epoch	Accuracy
model_1	SZDTROBERTA–7000	to characters	3	10,05%
model_2	SZDTROBERTA–7000	to words	3	36,15%
model_3	HIL_ROBERTA	to words and word parts	3	33,75%
model_4	HIL_ROBERTA	to words and word parts	6	47,05%
model_5	HIL_ROBERTA	to words and word parts	12	74,10%

A kiértékelés összefoglalása: SZDT–10000 tanítóminta, egyetlen szándék

A modellek manuális tesztelése is megtörtént néhány, a tanítómintában nem szereplő mondattal. Ebben az esetben is a model_5 lett a legeredményesebb.

a) Kirándulni mentünk.
b) Nagyon érdekes és szép volt ez a dombok övezte hely.
c) A boltba vittünk néhány almát.
d) Mit gondolsz a világról?
e) Sajnálom, hogy így elromlott az autó!

	model_1	model_2	model_3	model_4	model_5
a)	elindultunk	mentünk	mentünk	mentünk	mentünk
b)	elindultunk	volt	volt	volt	volt
c)	elindultunk	van	volt	van	vittünk
d)	elindultunk	eljött	van	van	gondolok
e)	elindultunk	van	van	van	van

Az egyes modellek által megállapított szándék

2.2.2. NLU_BENCHMARK tanítómintán végzett kísérletek

Az angol nyelvű modell manuális kiértékelése néhány saját mondat segítségével szintén megtörtént. Az eredmény a várakozásoknak megfelelő lett.

Utterance	Detected intent
Is it sunny today?	*GetWeather*
I would like to hear a nice song!	*PlayMusic*
Is it sunny today? I would like to hear a nice song!	*BookRestaurant*

Angol mondatok és a megállapított szándék

2.2.3. Magyar nyelvre automatikusan fordított NLU_BENCHMARK_HU_RAW tanítómintán végzett kísérletek

A tanítást 3, illetve 6 epochon keresztül végeztük, először az SZDTROBERTA–7000 alapmodell, majd a HIL_ROBERTA alapmodell finomhangolásával. Célunk az volt, hogy a két modell összehasonlítható legyen a szándékdetektálási feladatok megoldása szempontjából.

model_1:
SZDTROBERTA–7000 tokenizer and model, epoch 3, Accuracy: 91,9%
model_2:
HIL_ROBERTA tokenizer and model, epoch 3, Accuracy: 94,52%

model_3:
SZDTROBERTA–7000 tokenizer and model, epoch 6, Accuracy: 97,14%
model_4:
HIL_ROBERTA tokenizer and model, epoch 6, Accuracy: 97,61%

A modellek manuális kiértékelése

AddToPlaylist:
1. Tedd a listámra a *Sose halunk meg* című számot.
2. Add a kedvencekhez Michael Jacksontól a Billie Jeant.

BookRestaurant:
3. Ma estére foglalj asztalt az ablak mellé.
4. Ma ebédelni szeretnék valahol.

GetWeather:

5. Milyen idő lesz holnap reggel?
6. Elmehetek vajon sétálni az este?

PlayMusic:

7. Játszd le a kedvenc számomat!
8. Jó lenne hallani valami zenét.

RateBook:

9. A Harry Potter-könyveknek adj egy kilences értékelést.
10. A reggel olvasott könyvem nagyon gyenge.

SearchCreativeWork:

11. Hol található Michelangelo leghíresebb szobra, a David?
12. Keresd meg nekem a *Nyomorultak* című művet!

SearchScreeningEvent:

13. Milyen filmeket adnak a mozik?
14. Hol láthatok egy jó vígjátékot?

	model_1	**model_2**	**model_3**	**model_4**
1	AddToPlaylist	AddToPlaylist	AddToPlaylist	AddToPlaylist
2	SearchScreeningEvent	GetWeather	SearchScreeningEvent	SearchCreativeWork
3	PlayMusic	BookRestaurant	BookRestaurant	BookRestaurant
4	PlayMusic	GetWeather	GetWeather	BookRestaurant
5	GetWeather	GetWeather	GetWeather	GetWeather
6	SearchScreeningEvent	GetWeather	GetWeather	SearchCreativeWork
7	PlayMusic	PlayMusic	PlayMusic	PlayMusic
8	PlayMusic	PlayMusic	PlayMusic	PlayMusic
9	RateBook	RateBook	BookRestaurant	SearchCreativeWork
10	RateBook	BookRestaurant	SearchScreeningEvent	SearchCreativeWork
11	SearchScreeningEvent	GetWeather	BookRestaurant	SearchScreeningEvent
12	SearchCreativeWork	SearchCreativeWork	SearchCreativeWork	SearchCreativeWork
13	SearchScreeningEvent	SearchScreeningEvent	SearchScreeningEvent	SearchScreeningEvent
14	SearchScreeningEvent	SearchScreeningEvent	SearchScreeningEvent	SearchScreeningEvent

2.2.4. Az SZDTROBERTA modell tanításjellemzői, statisztikák:

***** Running training *****
Num examples = 81 967
Num Epochs = 1
Instantaneous batch size per device = 4
Total train batch size (w. parallel, distributed & accumulation) = 4
Gradient Accumulation steps = 1
Total optimization steps = 20 492

CPU times: user 32 min, sys: 6.74 s, total: 32 min 7 s
Wall time: 32 min 2 s

TrainOutput(global_step=20492, training_loss=7.5551160447140955, metrics={
'train_runtime': 1922.061,
'train_samples_per_second': 42.645,
'train_steps_per_second': 10.661,
'total_flos': 825760901437056.0,
'train_loss': 7.5551160447140955,
'epoch': 1.0})

2.2.5. Magyar nyelvre fordított tanítóminta manuális javítása után előállt NLU_BENCHMARK_HU tanítómintán végzett kísérletek

Az NLU_BENCHMARK_HU tanítómintával az SZDTROBERTA alapmodellt finomhangoltuk.
A harmadik epoch végére elért Loss, illetve Accuracy:
Loss: 0.0303784366697073. Accuracy: 97,14%

A hatodik epoch végére elért Loss, illetve Accuracy:
Loss: 0.0008110094931907952. Accuracy: 98,01%

A HIL_ROBERTA alapmodellen is elvégeztük ugyanezen műveletsort.
A harmadik epoch végére elért Loss, illetve Accuracy:
Loss: 0.04047880321741104. Accuracy: 96,66666666666667%

A hatodik epoch végére elért Loss, illetve Accuracy:
Loss: 0.00409882515668869. Accuracy: 99,76190476190476%

A modellek kódjai, és eredményei:

model_1:
SZDTROBERTA tokenizer and model, epoch: 3, Accuracy: 97,1%

model_2:
SZDTROBERTA tokenizer and model, epoch: 6, Accuracy: 98,1%

model_3:
HIL_ROBERTA tokenizer and model, epoch: 3, Accuracy: 96,7%

model_4:
HIL_ROBERTA tokenizer and model, epoch: 6, Accuracy: 99,8%

A már korábban megadott 14 saját mondattal elvégeztük a manuális tesztelést mind a négy modell felhasználásával. A hibás predikciók száma csökkent.

	model_1	**model_2**	**model_3**	**model_4**
1	AddToPlaylist	AddToPlaylist	AddToPlaylist	AddToPlaylist
2	SearchScreeningEvent	SearchScreeningEvent	GetWeather	SearchScreeningEvent
3	BookRestaurant	BookRestaurant	BookRestaurant	BookRestaurant
4	BookRestaurant	BookRestaurant	GetWeather	GetWeather
5	GetWeather	GetWeather	GetWeather	GetWeather
6	GetWeather	RateBook	GetWeather	GetWeather
7	PlayMusic	PlayMusic	PlayMusic	PlayMusic
8	PlayMusic	PlayMusic	PlayMusic	PlayMusic
9	SearchCreativeWork	RateBook	RateBook	RateBook
10	SearchCreativeWork	RateBook	RateBook	RateBook
11	SearchScreeningEvent	SearchScreeningEvent	GetWeather	SearchScreeningEvent
12	SearchCreativeWork	SearchCreativeWork	SearchCreativeWork	SearchCreativeWork
13	SearchScreeningEvent	SearchScreeningEvent	SearchScreeningEvent	SearchScreeningEvent
14	SearchScreeningEvent	SearchScreeningEvent	GetWeather	SearchScreeningEvent

3. Technológiai folyamatok bemutatása

Technológiai folyamat szöveges ismertetése

Magyar SZDT–10000 tanítóminta feldolgozásának finomhangolási konfigurációja. Alapmodell: SZDTROBERTA–7000


```
{
 "architectures": [
   "RobertaForMaskedLM"
 ],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "classifier_dropout": null,
 "eos_token_id": 2,
 "finetuning_task": null,
 "gradient_checkpointing": false,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "initializer_range": 0.02,
 "intermediate_size": 3072,
 "layer_norm_eps": 1e-12,
 "max_position_embeddings": 514,
 "model_type": "roberta",
 "num_attention_heads": 12,
 "num_hidden_layers": 6,
 "num_labels": 2924,
 "output_attentions": false,
 "output_hidden_states": false,
 "pad_token_id": 1,
 "position_embedding_type": "absolute",
 "pruned_heads": {},
 "torch_dtype": "float32",
 "torchscript": false,
 "transformers_version": "4.11.0.dev0",
 "type_vocab_size": 1,
 "use_cache": true,
 "vocab_size": 52000
}
```

Angol nyelvű NLU-BENCHMARK tanítóminta finomhangolási konfigurációja. Alapmodell: ROBERTA_BASE


```
{
 "architectures": [
   "RobertaForMaskedLM"
 ],
 "attention_probs_dropout_prob": 0.1,
 "bos_token_id": 0,
 "eos_token_id": 2,
 "finetuning_task": null,
 "hidden_act": "gelu",
 "hidden_dropout_prob": 0.1,
 "hidden_size": 768,
 "initializer_range": 0.02,
 "intermediate_size": 3072,
 "layer_norm_eps": 1e-05,
 "max_position_embeddings": 514,
 "model_type": "roberta",
 "num_attention_heads": 12,
 "num_hidden_layers": 12,
 "num_labels": 7,
 "output_attentions": false,
 "output_hidden_states": false,
 "pad_token_id": 1,
 "pruned_heads": {},
 "torchscript": false,
 "type_vocab_size": 1,
 "vocab_size": 50265
}
```

Magyar nyelvre automatikusan fordított NLU-BENCHMARK_HU_1 tanítóminta finomhangolási konfigurációja.
Alapmodell: SZDTROBERTA–7000


```
{
"architectures": [
"RobertaForMaskedLM"
],
"attention_probs_dropout_prob": 0.1,
"bos_token_id": 0,
"classifier_dropout": null,
"eos_token_id": 2,
"finetuning_task": null,
"gradient_checkpointing": false,
"hidden_act": "gelu",
"hidden_dropout_prob": 0.1,
"hidden_size": 768,
"initializer_range": 0.02,
"intermediate_size": 3072,
"layer_norm_eps": 1e-12,
"max_position_embeddings": 514,
"model_type": "roberta",
"num_attention_heads": 12,
"num_hidden_layers": 6,
"num_labels": 7,
"output_attentions": false,
"output_hidden_states": false,
"pad_token_id": 1,
"position_embedding_type": "absolute",
"pruned_heads": {},
"torch_dtype": "float32",
"torchscript": false,
"transformers_version": "4.11.0.dev0",
"type_vocab_size": 1,
"use_cache": true,
"vocab_size": 52000
}
```

Magyar nyelvre automatikusan fordított NLU-BENCHMARK_HU_1 tanítóminta finomhangolási konfigurációja alapmodell: HIL_ROBERTA


```
{
"architectures": [
"RobertaForMaskedLM"
],
"attention_probs_dropout_prob": 0.1,
"bos_token_id": 0,
"eos_token_id": 2,
"finetuning_task": null,
"gradient_checkpointing": false,
"hidden_act": "gelu",
"hidden_dropout_prob": 0.1,
"hidden_size": 768,
"initializer_range": 0.02,
"intermediate_size": 3072,
"layer_norm_eps": 1e-12,
"max_position_embeddings": 514,
"model_type": "roberta",
"num_attention_heads": 12,
"num_hidden_layers": 6,
"num_labels": 7,
"output_attentions": false,
"output_hidden_states": false,
"pad_token_id": 1,
"pruned_heads": {},
"torchscript": false,
"type_vocab_size": 1,
"vocab_size": 30000
}
```

Az SZDTROBERTA alapmodell konfigurációja


```
{
  "attention_probs_dropout_prob": 0.1,
  "bos_token_id": 0,
  "classifier_dropout": null,
  "eos_token_id": 2,
  "hidden_act": "gelu",
  "hidden_dropout_prob": 0.1,
  "hidden_size": 768,
  "initializer_range": 0.02,
  "intermediate_size": 3072,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 514,
  "model_type": "roberta",
  "num_attention_heads": 12,
  "num_hidden_layers": 6,
  "pad_token_id": 1,
  "position_embedding_type": "absolute",
  "transformers_version": "4.14.0.dev0",
  "type_vocab_size": 1,
  "use_cache": true,
  "vocab_size": 52000
}
```

A javított magyar NLU-BENCHMARK_HU tanítóminta finomhangolási konfigurációja. Alapmodell: SZDTROBERTA


```
{
"architectures": [
"RobertaForMaskedLM"
],
"attention_probs_dropout_prob": 0.1,
"bos_token_id": 0,
"classifier_dropout": null,
"eos_token_id": 2,
"finetuning_task": null,
"hidden_act": "gelu",
"hidden_dropout_prob": 0.1,
"hidden_size": 768,
"initializer_range": 0.02,
"intermediate_size": 3072,
"layer_norm_eps": 1e-12,
"max_position_embeddings": 514,
"model_type": "roberta",
"num_attention_heads": 12,
"num_hidden_layers": 6,
"num_labels": 7,
"output_attentions": false,
"output_hidden_states": false,
"pad_token_id": 1,
"position_embedding_type": "absolute",
"pruned_heads": {},
"torch_dtype": "float32",
"torchscript": false,
"transformers_version": "4.14.0.dev0",
"type_vocab_size": 1,
"use_cache": true,
"vocab_size": 52000
}
```

A javított magyar NLU-BENCHMARK_HU tanítóminta finomhangolási konfigurációja. Alapmodell: HIL_ROBERTA


```
{
"architectures": [
"RobertaForMaskedLM"
],
"attention_probs_dropout_prob": 0.1,
"bos_token_id": 0,
"eos_token_id": 2,
"finetuning_task": null,
"gradient_checkpointing": false,
"hidden_act": "gelu",
"hidden_dropout_prob": 0.1,
"hidden_size": 768,
"initializer_range": 0.02,
"intermediate_size": 3072,
"layer_norm_eps": 1e-12,
"max_position_embeddings": 514,
"model_type": "roberta",
"num_attention_heads": 12,
"num_hidden_layers": 6,
"num_labels": 7,
"output_attentions": false,
"output_hidden_states": false,
"pad_token_id": 1,
"pruned_heads": {},
"torchscript": false,
"type_vocab_size": 1,
"vocab_size": 30000
}
```


4. Összegzés

Létrehoztuk a magyar nyelvű NLU_BENCHMARK_HU tanítómintát. Az automatikus fordítás után keletkezett tanítóminta karakterkódolási problémáit kijavítottuk. Az előállt tanítófájl nyelvhelyességét ezután manuálisan sorról sorra kijavítottuk. Így a fájl magyar nyelvű szövegosztályozó, szándékmeghatározó algoritmusok tesztelésére vált alkalmassá.

Elkészítettük az előtanított SZDTROBERTA alapmodellt. Mind az SZDTROBERTA, mind a Nyelvtudományi Intézet HIL_ROBERTA modelljén kísérleteket futtattunk, és összevetettük az eredményeket.

Az eddigi tapasztalatok alapján szükséges megállapítani hogyan kell a meglévő tanítómintát úgy módosítani, hogy a többszörös szándékfeltárás finomhangolása elvégezhető legyen. A finomhangoláshoz szükségünk van egy annotációkészítő alkalmazásra. Kiválasztottunk és elemeztünk néhány ilyen eszközt.

Elemeztük, hogy lehetséges-e a meglévő modellstruktúrával kapcsolt többszörös szándékfeltárást, illetve szekvenciális címkézést végeznünk. A tapasztalat szerint tovább kell fejlesztenünk a jelenlegi megoldást, ennek mikéntjét meghatároztuk, illetve definiáltuk milyen tanítóhalmaz kiegészítésre van szükségünk.

APPLICATION STUDIES OF AUTOENCODERS

SAMAD DADVANDIPOUR

At this stage of the research we have illustrated two applications of autoencoders, with respect medical and MNIST datasets in case of cancer and handwritten, where the aim is to detect the applications Malignant or Benign.

1. Introduction

Autoencoders and their application in ANN

Autoencoder neural networks are unsupervised machine learning algorithms. They apply backpropagation, setting the target values equal to the inputs. Thus, they are algorithms similar to PCA but minimize the same objective function. An autoencoder is a neural network whose target output is its input. Autoencoders are pretty identical to PCA, but they are more flexible when compared to the others. For example, autoencoders can represent linear and no-linear transformations in encoding, but PCA can perform the linear transformation.

Need for autoencoders

Data compression is a big topic used in computer vision, computer networks, and many other applications. Now the point of the data compression is to convert input into smaller representation data that we recreated to a degree of quality. The small representation would be passed around, and when anyone needed the original, they would reconstruct from the smaller representation.

An encoder also gives a representation as to the output of each layer, and having multiple representations of different dimensions is always useful. So an autoencoder could tell you make use of pre-trained layers from another model to apply transfer to prime the encoder or the decoder; even though practical applications of autoencoders were pretty rare sometimes back today, data denoising and dimensionality reduction for data visualization are considered as two main interesting practical applications of autoencoders. With appropriate dimensionality sparsing constraints, autoencoders can learn data projection.

Descriptions

There are basically three main layers: the encoder, the coder, and the encoder. The first component that is the encoder, is the part of the neural network that compresses the input into latent space representation in a reduced dimension. The compressed image typically looks garbled/distorted/mixed up, nothing like the original image. The next component represents the latent space. Code is the

part of the network that represents the compressed input fed to the decoder. The third component is the decoder., which decodes the encoded image back to the original dimension with close or same images. In fact, the decoded image is a lossy reconstruction of the original image. It reconstructs the input from the latent space representation.

The properties of autoencoders

They can only compress data similar to what they have been trained on and autoencoders that have been trained. Autoencoders are usually called lossy. It means that the decompressed outputs are degraded contrasted to the original inputs. Now, if you have appropriate training data, it is easy to train specialized algorithms to work well on a particular input type.

Training autoencoders

It doesn't need any new engineering; furthermore, in almost all contents where the autoencoder is used, the compression and decompression functions are carried out with the neural network [17, 18] [19, 21].

Preprocessing before training

There are four hyperparameters that we need to set before training them.

1. The first one is the code size

The code size represents the number of nodes in the middle layer smaller size results in more compression.

2. The second parameter is the number of layers.

Now the autoencoder can be as we want it to be. We may have two or more layers in both the encoders and decoders without considering the input and outputs. Next is the loss function, so we either use mean squared error or binary cross-entropy. Now, if the input values are in the range of 0 to 1, then we typically use cross-entropy. Otherwise, we use the mean squared error.

3. The third parameter is the nodes number for each layer.

The number of nodes for each layer decreases with each subsequent layer of the encoder and increases back in the encoder [18]. Also, the decoder is symmetric

in terms of the layer structure, but this is unnecessary, and we have total control over this parameter. The architecture of an autoencoder has a deeper insight with a couple of layers between the input and output. The sizes of these layers are smaller than the input layer.

Case study 1: Medical application of Autoencoder

Problem statement: Given the prostate cancer dataset, try to predict whether the cancer is Malignant or Benign [5].

2. Summary of research results

Using the diagnosis and statistical dataset for prostate cancer**, (**Table 1 and 2), we try to predict whether the cancer is Malignant or Benign. The application is using the tensor flow matrix in this respect. Also Keras model used for the dimensional reduction.

Table 1. Dataset

id	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
4	M	14	16	78	386	0.07	0.284	0.26	0.096999999 99999999
1	M	23	12	151	954	0.1430000 00000000 02	0.278	0.242	0.079
3	M	21	27	130	1203	0.125	0.16	0.207	0.06
5	M	9	19	135	1297	0.141	0.133	0.181000000 00000002	0.059000000 000000004
2	B	9	13	133	1326	0.1430000 00000000 02	0.079	0.181000000 00000002	0.057

As is shown in the above table, we have eight independent features and one dependent feature (Diagnosis result), which are the number of features and the description of the dataset [4].

2.1. Required libraries:

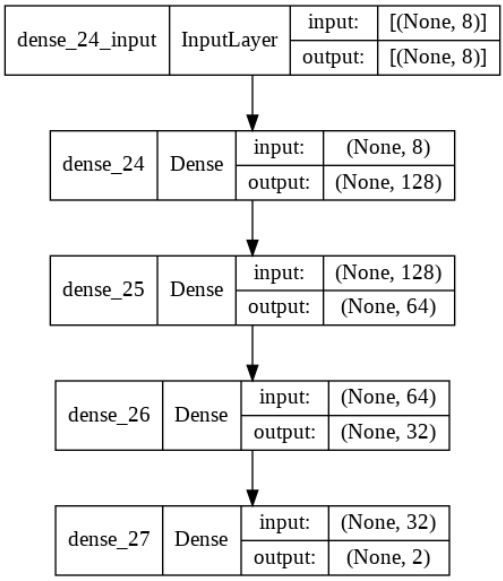
```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Model, Sequential
from tensorflow.keras.layers import Dense, Flatten, Reshape
import pandas as pd
import random
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import seaborn as sns
```

Table. 2. *Some statistical results of the dataset*

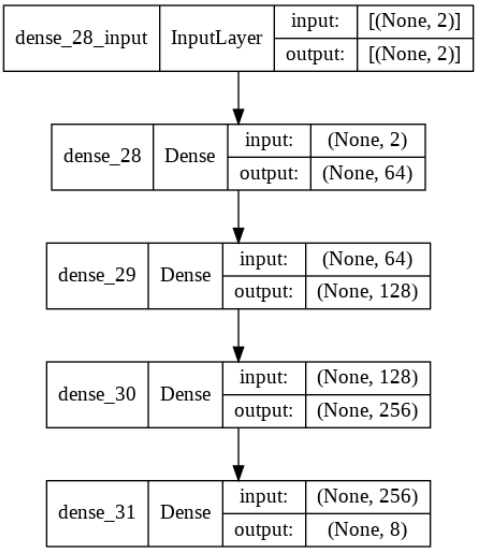
index	id	diagnosis_result	radius	texture	perimeter	area	smoothness	compactness	symmetry	fractal_dimension
count	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
mean	50.5	0.62	16.85	18.23	96.78	702.88	0.10273	0.1267	0.19317	0.064690 00000000 001
std	29.011 491975 882016	0.4878317 31214563 25	4.8790 937227 68149	5.1929 536563 27742	23.676 088606 80268	319.71 089465 580644	0.014641 7522547 9892	0.06114356 346775475	0.03078 503342 256237 7	0.008150 96821416 2219
min	1.0	0.0	9.0	11.0	52.0	202.0	0.07	0.038	0.135	0.053
25%	25.75	0.0	12.0	14.0	82.5	476.75	0.0935	0.0805	0.172	0.059000 00000000 0004
50%	50.5	1.0	17.0	17.5	94.0	644.0	0.102	0.1185	0.19	0.063
75%	75.25	1.0	21.0	22.25	114.25	917.0	0.111999 9999999 9999	0.157	0.209	0.069
max	100.0	1.0	25.0	27.0	172.0	1878.0	0.143000 0000000 0002	0.345	0.304	0.096999 99999999 999

We build an encoder to reduce the number of features from 8 to 2 and check whether we can get the same results. Model architecture for encoder unit, dimensions of original space = 8, latent space dimensions = 2.

2.2. The model architecture

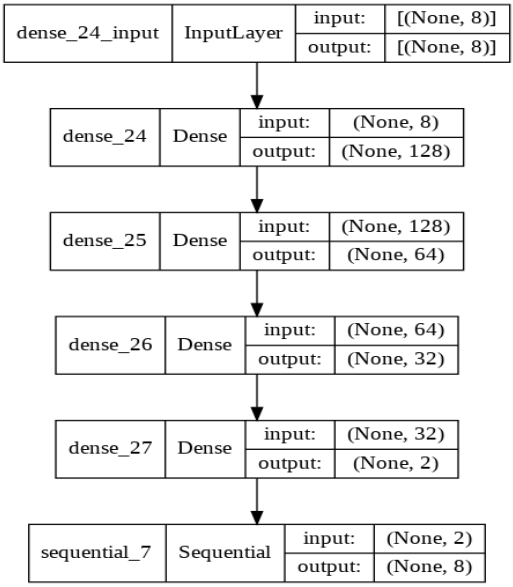


Decoder unit architecture: Input dimensions = 2, reconstructed dimensions = 8



Autoencoder architecture: Combining both encoder and decoder so, in this model, both original dimensions and output or reconstructed dimensions should be the same [1, 2, 3].

```
# This is the dimension of the original space
input_dim = 8

# This is the dimension of the latent space (encoding space)
latent_dim = 2

encoder = Sequential([
   Dense(128, activation='relu', input_shape=(input_dim,)),
   Dense(64, activation='relu'),
   Dense(32, activation='relu'),
   Dense(latent_dim, activation='relu'),
   #Dense(1, activation = 'sigmoid')
])

decoder = Sequential([
Dense(64, activation='relu', input_shape=(latent_dim,)),
   Dense(128, activation='relu'),
   Dense(256, activation='relu'),
   Dense(input_dim, activation=None)
])

autoencoder = Model(inputs=encoder.input, outputs=decoder(encoder.output))
autoencoder.compile(loss='mse', optimizer='adam')
```



Our autoencoder is still untrained at this time. So let's try feeding it some instances from the dataset to see how effectively it starts to reconstruct the following:

```
def plot_orig_vs_recon(title='', n_samples=3):
    fig = plt.figure(figsize=(10,6))
    plt.suptitle(title)
    for i in range(3):
        plt.subplot(3, 1, i+1)

   idx = random.sample(range(x_train.shape[0]), 1)
        plt.plot(autoencoder.predict(x_train[idx]).squeeze(), label='reconstructed' if i ==
0 else '')
        plt.plot(x_train[idx].squeeze(), label='original' if i == 0 else '')
        fig.axes[i].set_xticklabels(metric_names)
        plt.xticks(np.arange(0, 10, 1))
        plt.grid(True)
        if i == 0: plt.legend();

plot_orig_vs_recon('Before training the Autoencoder')
```

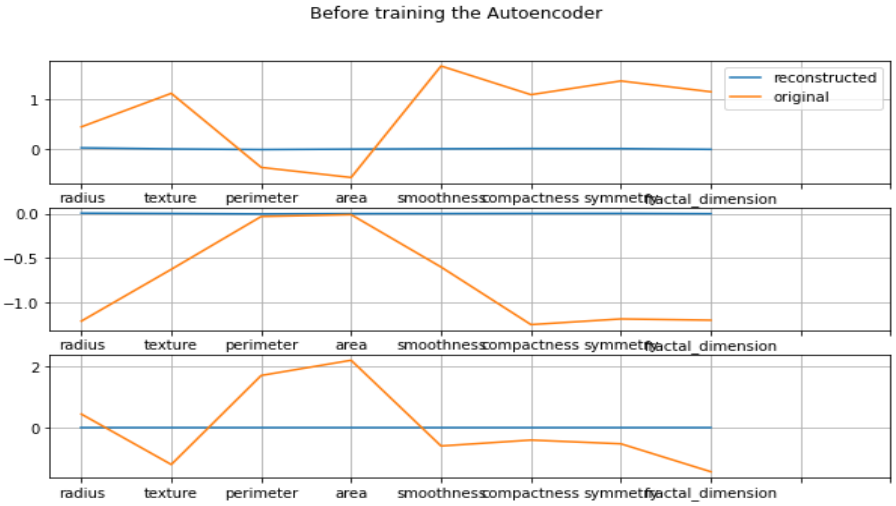


Figure 1. *Illustration of original data taken from the prostate*

As we can depict from the above graph, there is a lot of difference between the original and reconstructed data. We say that both original and reconstructed data are similar if the graph lines overlap. Now we will train our autoencoder with the following parameters [2, 3]:

1. Loss = MSE
2. Optimizer = Adam
3. Epochs = 5000 and
4. Batch size = 32

After training the autoencoder, the loss that occurred is given as:

```
#After training the encoder
model_history = autoencoder.fit(x_train, x_train, validation_data = (x_test, x_te
st), epochs=5000, batch_size=32, verbose=0)

plt.plot(model_history.history["loss"])
plt.title("Loss vs. Epoch")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.grid(True)
plt.savefig("loss vs Epochs")
```

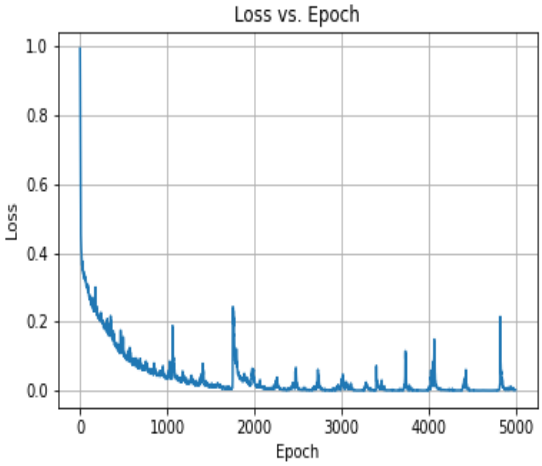


Figure 2. *training result based on given data, based on loss Vs. Given epoch*

Our model converged. It worked, and now we check the reconstruction on a trained autoencoder:

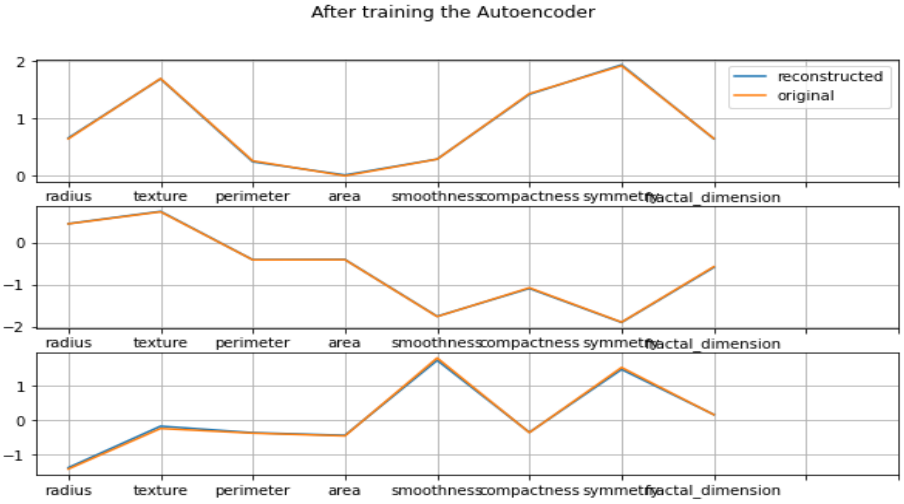


Figure 3. *Illustration of overlapping the original data after training, the training detects the given cancerous data*

3. Conclusion

In the digital era, millions of bytes of data are exchanged every day on the internet. Although storing and analyzing this humongous data is challenging, it also

reduces energy consumption. Autoencoder is a special unsupervised or specifically self-supervised neural network consisting of an encoder and decoder unit. First, it starts widespread, then its units/connections are pushed closer to the center and spread out again. This architecture makes the autoencoder compress the training data's informational content, encoding it in a low-dimensional space. In the case of the prostate in general, a similar process happens. Still, when the prostate is under tumor type, ML can diagnose whether the cancer is Malignant (bad-natured) or Benign (good-natured). We used an autoencoder model similar to the prostate with given data for both cases in this example. After considering the critical eight features by the autoencoder, the original and reconstructed lines overlap almost completely, which tells us our model worked perfectly, i.e., it reconstructed the input data of 8 dimensions to the reconstructed output of 8 dimensions using the latent space of two sizes. As a result, the reconstructed values are almost identical to the originals!

Let us now examine the latent space. We'll use a 2D scatterplot to visualize it because it's two-dimensional. The 8-dimensional data is projected onto a plane in this way.

```
encoded_x_train = encoder(x_train)
plt.figure(figsize=(6,6))
plt.scatter(encoded_x_train[:, 0], encoded_x_train[:, 1], alpha=.8)
plt.xlabel('Latent Dimension 1')
plt.ylabel('Latent Dimension 2');
```

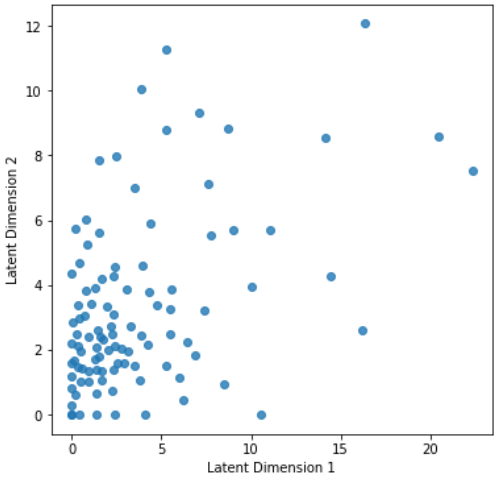


Figure 4. *Illustration of data dimensions*

The decoder needs this 2D imprint to reconstruct the original 8-dimensional space. However, there are a lot of points crammed into the bottom left corner. This is because we want each class's data points to create unique clusters in a classification situation.

Case study 2: Handwritten Application of Autoencoder

In the MNIST dataset there are 60.000 examples in the training set and 10.000 samples in the test set. An image of a handwritten 28x28 grayscale digit from 0 to 9 is used in each case. We preprocessed the input data to be presentable for the encoder model [16, 17] [9, 10].

4. Summary of research results

4.1. Application

Input data:

```
# Load the MNIST data set
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

#Normalize pixel values to [0., 1.]
x_train = x_train / 255.
x_test = x_test / 255.

# Take a look at the dataset
n_samples = 10
idx = random.sample(range(x_train.shape[0]), n_samples)
plt.figure(figsize=(15,4))
for i in range(n_samples):
plt.subplot(1, n_samples, i+1)
plt.imshow(x_train[idx[i]].squeeze());
plt.xticks([], [])
    plt.yticks([], [])
```

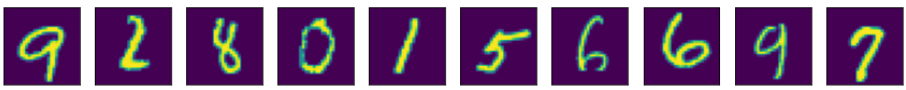
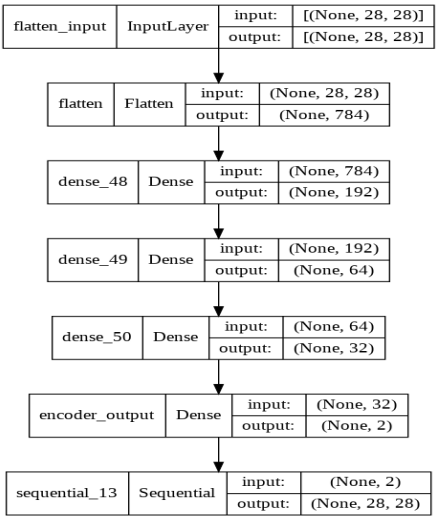


Figure 5. *Number of the sample were taken for training (15 cm × 4 cm)*

Autoencoder encoder architecture, image size = 28 * 28 and dimension of latent space = 2.

```
from keras.regularizers import*
from keras.layers import*

# This is the dimension of the latent space (encoding space)
latent_dim = 2

# Images are 28 by 28
img_shape = (x_train.shape[1], x_train.shape[2])

encoder = Sequential([
Flatten(input_shape=img_shape),
Dense(192, activation='sigmoid'),
Dropout(0.3),
Dense(64, activation='sigmoid'),
Dropout(0.4),

Dense(32, activation='sigmoid'),

Dense (latent_dim, name='encoder_output')
])

decoder = Sequential([
Dense(64, activation='sigmoid', input_shape=(latent_dim,)),
Dropout(0.3),
Dense(128, activation='sigmoid'),
Dropout(0.4),

Dense(img_shape[0] * img_shape[1], activation='relu'),

Reshape(img_shape)

])
```

4.2. The applied Model



We'll design a custom callback by subclassing the tf.Keras.callback.Callback. To visualize the autoencoder, how it builds up the latent space representation as we train it. We will override the function or method on_epoch_begin (self, epoch, logs = None), which is called at the start of the epoch during the training phase; we will extract the representation of the latent space by looking up in the code and plotting it. There is a method in the Keras layer to get the output of an intermediate layer-output. We will train our model on the following hyperparameters [7, 15] [16, 17]:

1. Loss = Binary cross entropy.
2. Optimizer = Adam.
3. Epochs = 12 (number where each latent represented image trains the 32 given dataset pictures) for the decoding parts.
4. Batch size = 32 the sample pictures.

The evolution of latent space representation as the autoencoder is trained, starting at the top left with an untrained state and finishing with a wholly trained state at the bottom right. All of the original space data is projected on the same point of the latent space before the first epoch. However, when the autoencoder learns, the points associated with various classes begin to decouple.

```
class TestEncoder(tf.keras.callbacks.Callback):

def __init__(self, x_test, y_test):

super(TestEncoder, self).__init__()
self.x_test = x_test
self.y_test = y_test
self.current_epoch = 0

def on_epoch_begin(self, epoch, logs={}):
self.current_epoch = self.current_epoch + 1
encoder_model = Model(inputs=self.model.input,
outputs=self.model.get_layer('encoder_output').output)
encoder_output = encoder_model(self.x_test)
plt.subplot(4, 3, self.current_epoch)
plt.scatter(encoder_output[:, 0],
encoder_output[:, 1], s=20, alpha=0.8,
cmap='Set1', c=self.y_test[0:x_test.shape[0]])
plt.xlim(-9, 9)
plt.ylim(-9, 9)
plt.xlabel('Latent Dimension 1')
plt.ylabel('Latent Dimension 2')

autoencoder = Model(inputs=encoder.input, outputs=decoder(encoder.output))
autoencoder.compile(loss='binary_crossentropy', optimizer='adam', metrics
= ['accuracy'])
plt.figure(figsize=(15, 15))
model_history = autoencoder.fit(x_train, x_train, epochs=12, batch_size=32
, verbose=0,
callbacks=[TestEncoder(x_test[0:500], y_test[0:500])])
```

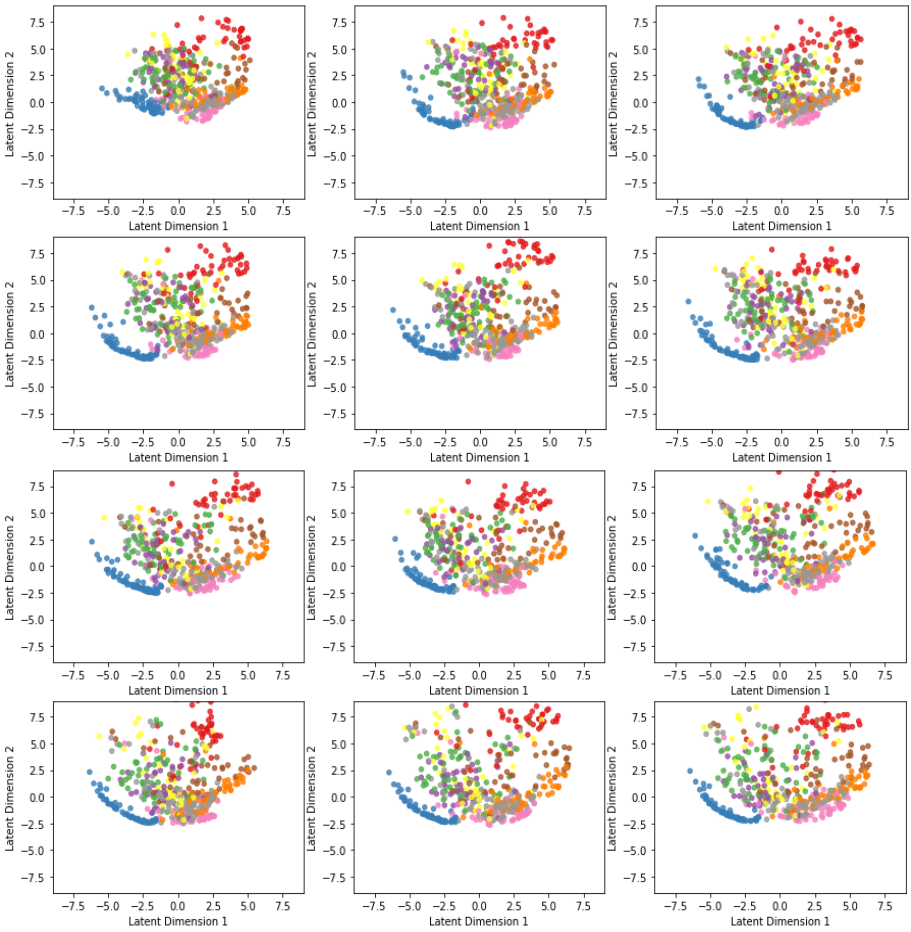


Figure 6. *The latent representation of MNIST*

Loss vs. epochs: to check whether our model converges or not.

```
plt.plot(model_history.history["loss"])
plt.title("Loss vs. Epoch")
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.grid(True)
```

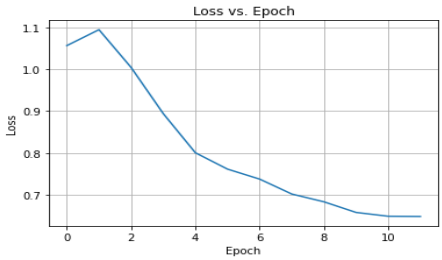
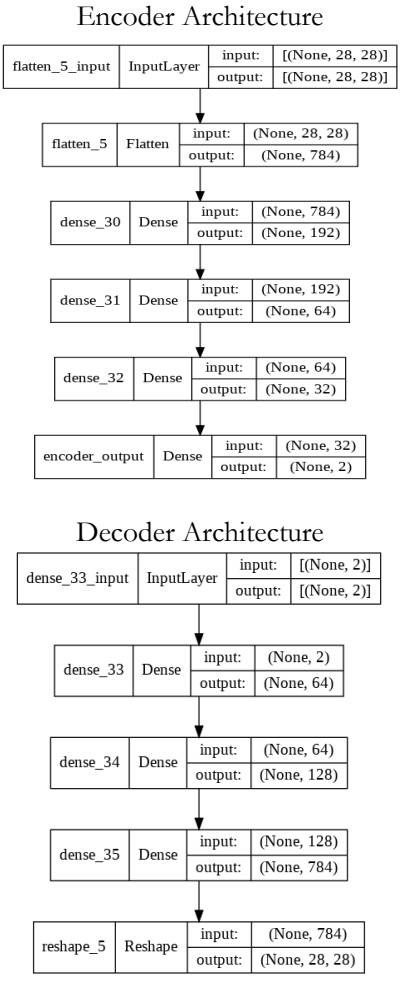


Figure 7. *Loss function, which shows a convergence of the model*



```
#plotting loss
plt.plot(history.history['loss'], label='Training data')
plt.plot(history.history['val_loss'], label='Validation data')
plt.title('Activity Loss')
plt.ylabel('Loss value')
plt.xlabel('No. epoch')
plt.legend(loc="upper left")
plt.show()
```

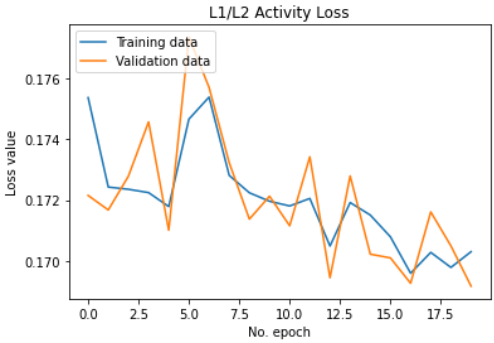


Figure 8. *Training Vs. Validation accuracy*

```
encoded_img = encoder.predict(x_test)
decoded_img = decoder.predict(encoded_img)
plt.figure(figsize=(10, 6))
for i in range(5):
    # Display original
    ax = plt.subplot(2, 5, (1,2))
    plt.imshow(x_test[i].reshape(28, 28))
    plt.show()
    #ax.get_xaxis().set_visible(False)
    #ax.get_yaxis().set_visible(False)
      # Display reconstruction
    ax = plt.subplot(2, 5, (1,2))
    plt.imshow(decoded_img[i].reshape(28, 28))
    plt.show()
    #ax.get_xaxis().set_visible(False)
    #ax.get_yaxis().set_visible(False)
plt.show()
```

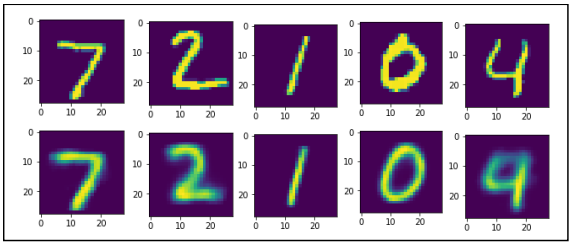


Figure 9. *Original Validation Vs. reconstructed data*

4.3. Autoencoder as Generative model using MNIS dataset

We use an autoencoder as a generative model. We can say when the autoencoder constructs a latent representation of the input data set, we use it as input. Then the decoder takes a sample of a random point of the latent space and produces a synthetic (fake) image [10, 11 and 8], [13, 14 and 12]. For example:

```
n_samples = 40
fake_sample = np.random.uniform(low=-20, high=20, size=(n_samples, 2))
plt.figure(figsize=(15,5))
for i in range(n_samples):
plt.subplot(4, n_samples//4, i+1)
fake_encoding = np.array([fake_sample[i]])
fake_digit = decoder(fake_encoding).numpy().squeeze()
plt.imshow(fake_digit);
plt.xticks([], [])
plt.yticks([], [])
```

We may sample a random point of the latent space of the autoencoder model and use it as input to the decoder to build a synthetic (fake) image. Once the autoencoder has generated a latent representation of the input data set. As an example,

Number of samples to generate = 40

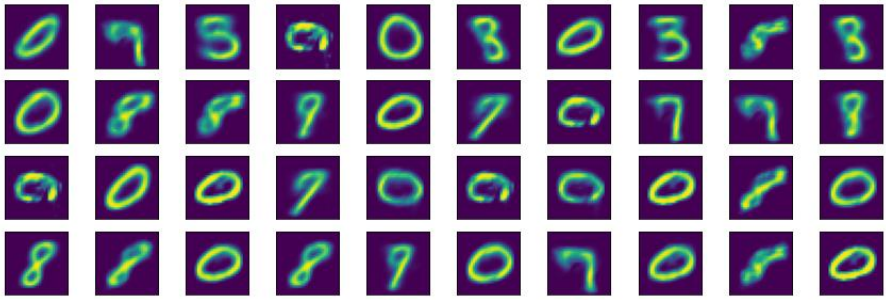


Figure 10. *Synthetic or fake images generated by the trained autoencoder*

5. Conclusion

This study has been carried out for dimensionality reduction using autoencoders for handwritten detection using the MNIST dataset. There are 60,000 examples in the training set and 10,000 samples in the test set. 28 * 28 grayscale image of a handwritten digit from 0 to 9 is used in each case. We preprocessed the input data to be presentable for the encoder model. The input image is 28 * 28, so it is

converted to 784 inputs. The dense 192 activations sigmoid function is the number of inputs neurons in the first layer. The dropout is 0.3 used for optimization. The output of neurons whose value is less than 0.3 is not activated. The dense 64 activation sigmoid is the number of inputs in the second layer. The dropout 0.4 was used for optimization; the output of neurons whose value is less than 0.4 are not activated.

References

[1] Patel, M. I., Suthar, S. and Thakar, J. (2019). *Survey on image compression using machine learning and deep learning.* https://doi.org/10.1109/ICCS45141.2019.9065473

[2] Bai, H., Zhang, M., Liu, M., Wang, A. and Zhao, Y. (2014). *Two-stage multiview image compression using interview SIFT matching.* https://doi.org/10.1109/DCC.2014.14

[3] Zhang, Q., Liu, D. and Li, H. (2018). Deep network-based image coding for simultaneous compression and retrieval. https://doi.org/10.1109/ICIP.2017.8296312

[4] https://www.kaggle.com/sajidsaifi/prostate-cancer.

[5] Abbasi, A. A., Hussain, L., Awan, I. A., Abbasi, I., Majid, A., Nadeem, M. S. A., & Chaudhary, Q. A. (2020). Detecting prostate cancer using a deep learning convolution neural network with the transfer learning approach. *Cognitive Neurodynamics*, 14 (4), pp. 523–533.

[6] Iqbal, S., Siddiqui, G. F., Rehman, A., Hussain, L., Saba, T., Tariq, U., & Abbasi, A. A. (2021). *Prostate Cancer Detection Using Deep Learning and Traditional Techniques.* https://ieeexplore.ieee.org/document/9349466

[7] Rippel, O. and Bourdev, L. (2017). Real-time adaptive image compression. *Proceedings of the 34th International Conference an Machine Learning*, Sydney, Australia, PMLR 70. http://proceedings.mlr.press/v70/rippel17a/rippel17a.pdf

[8] Choi, Y., Kang, D., Hwang, J. J. and Rhee, K. H. (2018). *JPEG Compression Detection Based on Edge-Corner Features Using SVM.* https://doi.org/810.1109/MLDS.2017.25

[9] Makar, M., Chang, C. L., Chen, D., Tsai, S. S. and Girod, B. (2009). Compression of image patches for local feature extraction. https://doi.org/10.1109/ICASSP.2009.4959710

[10] Robinson, J. and Kecman, V. (2003). Combining support vector machine learning with the discrete cosine transform in image compression. *IEEE Trans. Neural Networks*, https://doi.org/10.1109/TNN.2003.813842.

[11] Liu, X. and Yang, J. (2018). *Fast and High Efficient Color Image Compression Using Machine Learning*. https://doi.org/10.1109/IMCEC.2018.8469518

[12] Toderici, G. et al. (2017). *Full resolution image compression with recurrent neural networks*. https://doi.org/10.1109/CVPR.2017.577

[13] Quijas, J. and Fuentes, O. (2014). *Removing JPEG blocking artifacts using machine learning*. https://doi.org/10.1109/SSIAI.2014.6806033

[14] Cavigelli, L., Hager, P. and Benini, L. (2017). *CAS-CNN: A deep convolutional neural network for image compression artifact suppression*. https://doi.org/1410.1109/IJCNN.2017.7965927

[15] https://en.wikipedia.org/wiki/MNIST_database

[16] https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/CallbackList

[17] *Autoencoders*. https://saivenkatsudarshanam1996.medium.com/autoencoders-e42dc45b7bd0

[18] *Applied Deep Learning*. https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[19] *Understand Autoencoders by implementing in TensorFlow*. https://iq.opengenus.org/implementing-autoencoder-tensorflow/

[20] *Building Autoencoders in Keras*. https://blog.keras.io/building-autoencoders-in-keras.html

[21] *Autoencoders*. https://towardsdatascience.com/autoencoders-bits-and-bytes-of-deep-learning eaba376f23ad/

ISBN 978-963-358-262-6

A kiadásért felelős:
a Miskolci Egyetem rektora: Prof. dr. Horváth Zita
Megjelent a Miskolci Egyetemi Kiadó gondozásában
A kiadó felelős vezetője: Szendi Attila
Szöveggondozás, tördelés, tipográfia, borítóterv: Gramantik Csilla
Korrektor: Juhász Zoltán
GÉIK, Általános Informatikai Intézeti Tanszék – 2022 – ME