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 klasszikus 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:

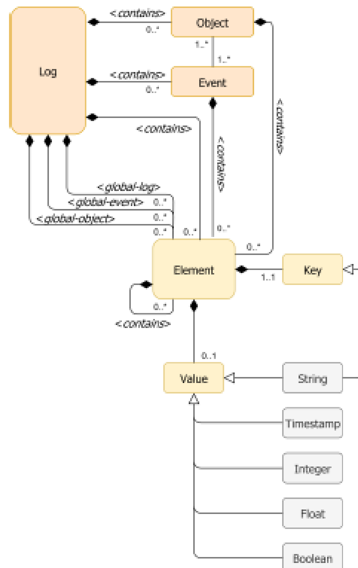
```
<events>
  <event>
    <string key="id" value="e1"/>
    <string key="activity" value="place_order"/>
    <date key="timestamp" value="2020-07-09T08:20:01.527+01:00"/>
    <list key="omap">
      <string key="object-id" value="i1"/>
      <string key="object-id" value="o1"/>
      <string key="object-id" 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.

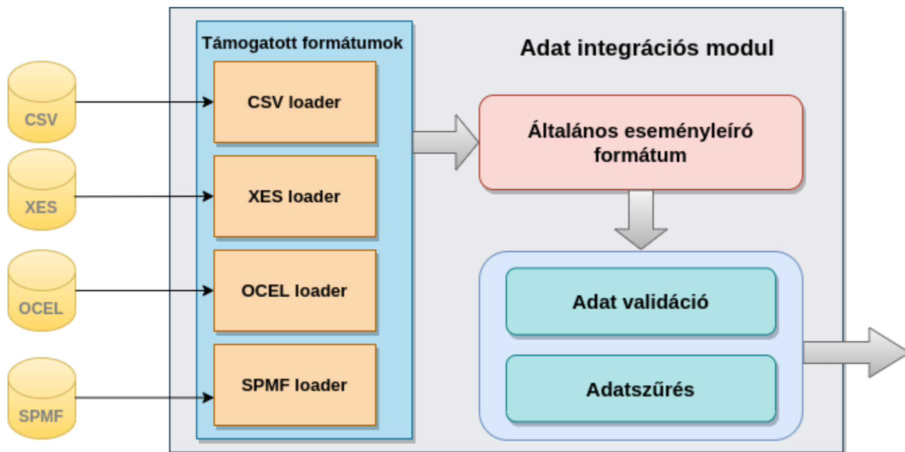
Alapvető probléma, hogy a különböző formátumok támogatása különböző, formátum-specifikus megvalósítást kíván meg a rendszertől. Fontos szempont, hogy külön kell választani az adatokat szolgáltató réteget, a predikciót végző logikai modultól. Bár elméletben megvalósítható az az irány, miszerint a neurális 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*

<u>Egyéb elvárt funkciók:</u>

<u>Eseményablakok támogatása:</u>

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

<u>Eseménysorok kiegészítése:</u>

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

<u>Eseménysorok átalakítása:</u>

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

<u>Eseményszekvenciák logikai ellenőrzése:</u>

- 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 letesztelésre 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.