ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS

INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Prokkály Dávid**
2025	Hallgató törzskönyvi száma:	**T/008579/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Biomatika és Alkalmazott Mesterséges Intelligencia Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Prokkály Dávid**
Törzskönyvi száma: T/008579/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

OCR Alapú Adatok Kezelése Generatív AI-val
Handling OCR-Based Data with Generative AI

Intézményi konzulensek: Dr. Eigner György, Szigeti Mark Bence
Külső konzulens:

Beadási határidő: 2024. december 15.

A záróvizsga tárgyai: Számítógép architektúrák
Mesterséges intelligencia specializáció

A feladat

Az OCR alapú adatok kezelése Generatív AI-val olyan lehetőségeket kínál, amelyek nagy mértékben elősegíthetik az adatfeldolgozást és az automatizálást. Ez a szakdolgozat egy olyan rendszert mutat be, amely már OCR által digitalizált adatokat használ fel bemenetként az AI modell számára. A program képes a kapott dokumentumok automatikus feldolgozására, elemzésére, valamint ezek alapján történő címkék generálására, így később egyszerű lekérdezésekkel is lehetséges lesz a dokumentumok keresése és kezelése. Ezáltal a felhasználó személy/cég produktivitása növelhető, hisz nem kell egy adott dokumentum keresésébe annyi időt és energiát fordítani.

A dolgozatnak tartalmaznia kell:

- OCR technológia,
- metaadatok kezelése,
- NLP,
- Generatív AI,
- értékelési metrikák és teljesítmény értékelés,
- Docker konténerizáció.

ÓBUDAI EGYETEM * Neumann János Informatikai Kar * 1. Budapest

Ph.

.....

Sájevicsné Dr. habil. Sápi Johanna
intézetigazgató

A szakdolgozat elévülésének határideje: **2026. december 15.**
(ÓE HKR 54.§ (10) bekezdés szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

Absztrakt

A dolgozat célja annak bemutatása, hogyan alkalmazható a generatív mesterséges intelligencia (AI) és a természetes nyelvfeldolgozás (NLP) az OCR (optikai karakterfelismerés) technológiával nyert szövegek kezelésére. Az OCR technológia lehetővé teszi a nyomtatott és kézzel írott szövegek digitalizálását, ám a kinyert szövegek gyakran pontatlanságokat és hibákat tartalmaznak, amelyek feldolgozása kihívást jelent.

A dolgozat részletes szakirodalmi áttekintést nyújt az OCR technológia működéséről és a metaadatok kezelésének fontosságáról. Bemutatja a generatív AI alapelveit és típusait, különös tekintettel a neurális hálózatok és mélytanulási algoritmusok szerepére a szöveggenerálás és javítás területén. Az NLP technikák segítségével lehetővé válik az OCR-ezett szövegek pontosabb értelmezése és feldolgozása, így növelve a digitális szövegek minőségét és használhatóságát.

A dolgozat továbbá foglalkozik a Docker konténerizációval, amely a fejlesztési folyamatokban játszik fontos szerepet, biztosítva a szoftverek hordozhatóságát és hatékony futtatását különböző környezetekben. Az automatizált és skálázható megoldások révén a generatív AI és az NLP technológiák hatékonyan integrálhatók a meglévő rendszerekbe.

Ez a szakdolgozat rávilágít arra, hogy a modern AI technológiák alkalmazása az OCR-ezett szövegek kezelésében jelentősen javíthatja a digitalizált adatok pontosságát és értékét, új lehetőségeket teremtve a dokumentumkezelés és az adatfeldolgozás területén.

Abstract

The purpose of this thesis is to demonstrate how generative artificial intelligence (AI) and natural language processing (NLP) can be applied to manage and improve texts obtained through optical character recognition (OCR) technology. OCR technology allows for the digitization of printed and handwritten texts, but the resulting texts often contain inaccuracies and errors that pose challenges for processing.

The thesis provides a comprehensive literature review on the operation of OCR technology and the importance of metadata management. It introduces the principles and types of generative AI, with a particular focus on the role of neural networks and deep learning algorithms in text generation and correction. Through the application of NLP techniques, it becomes possible to more accurately interpret and process OCR-scanned texts, thereby enhancing the quality and usability of digital texts.

Additionally, the thesis addresses Docker containerization, which plays a significant role in the development process by ensuring the portability and efficient execution of software in various environments. Automated and scalable solutions enable the effective integration of generative AI and NLP technologies into existing systems.

This thesis highlights how modern AI technologies, when applied to OCR-scanned text management, can significantly improve the accuracy and value of digitized data, creating new opportunities in document management and data processing.

Tartalomjegyzék

1. Bevezetés

Az optikai karakterfelismerés (OCR) technológiája jelentős fejlődésen ment keresztül az elmúlt években, lehetővé téve a nyomtatott és kézzel írott szövegek digitalizálását. Azonban az OCR folyamat eredménye gyakran tartalmaz pontatlanságokat és hibákat, amelyek manuális javítása és feldolgozása komoly kihívást jelent. A modern generatív mesterséges intelligencia (AI) és a természetes nyelvfeldolgozás (NLP) technikák ígéretes megoldást kínálnak ezen problémák enyhítésére és az OCR technológia által kinyert szövegek minőségének javítására.

A dolgozat célja, hogy bemutassa, miként alkalmazható a generatív AI és az NLP az OCR technológiával kinyert szövegek kezelésére és javítására. A generatív AI, különösen a neurális hálózatok és mélytanulási algoritmusok segítségével, képes új szövegeket generálni, illetve a meglévő szövegek hibáit kijavítani. Az NLP technikák lehetővé teszik az OCR-ezett szövegek pontosabb értelmezését és feldolgozását, növelve ezzel a digitalizált szövegek használhatóságát és minőségét.

A dolgozat keretében bemutatásra kerül a Docker konténerizáció is, amely a fejlesztési folyamatokban játszik fontos szerepet. A Docker konténerek biztosítják a szoftverek hordozhatóságát és hatékony futtatását különböző környezetekben, lehetővé téve a generatív AI és az NLP technológiák integrációját a meglévő rendszerekbe. Az automatizált és skálázható megoldások révén az OCR-ezett szövegek kezelése jelentősen javítható, új lehetőségeket teremtve a dokumentumkezelés és az adatfeldolgozás területén.

Ez a kutatás rávilágít arra, hogy a modern AI technológiák alkalmazása az OCR-ezett szövegek kezelésében jelentős előnyökkel járhat. Az OCR és a generatív AI/NLP technikák kombinációja révén lehetővé válik a szövegek pontosabb digitalizálása és javítása, ezáltal növelve azok pontosságát és értékét. A dolgozat részletesen bemutatja ezen technológiák működését, alkalmazási lehetőségeit és a fejlesztési folyamatokban betöltött szerepüket, hozzájárulva ezzel a digitalizált adatok hatékonyabb kezeléséhez és felhasználásához.

2. Szakirodalom feldolgozás

2.1 Az OCR technológia

2.1.1 Az OCR működése.

Az OCR (optikai karakterfelismerés) technológia a nyomtatott kézzel írott szövegek digitalizálását teszi lehetővé. Ennek lényege a karakterek felismerése és átalakítása gépileg olvasható formátumra. A rendszer általában hardver és szoftver együtteséből áll, például optikai szkenner (hardver) és karakterfelismerő szoftver, mely akár AI-t (mesterséges intelligencia) is alkalmazhat az intelligens karakterfelismeréshez. [1]

Az OCR szoftver a következő lépésekkel működik:

Képgyűjtés: A szkenner beolvassa a dokumentumokat, a szoftver elemzi a beolvasott képet. A világos területeket háttérnek, a sötét területeket pedig szövegnek minősíti.

Előfeldolgozás:

- Dőléskorrekció: A szkennelés során felmerülő igazítási problémák korrigálása, például a szkennelt dokumentum enyhe elforgatása.

- Binarizálás: Bináris képpé alakítjuk a színes képet. Ez a gyakorlatban egy köztes szürkeárnyalatos kép segítségével történik. Ezt a műveletet különböző módszerekkel lehet elvégezni, például:
    - Adaptív küszöbértékelés (Adaptive Thresholding)
    - Otsu-féle binarizáció
    - Helyi maximumok és minimumok módszere (Local Maxima and Minima Method)

- Zajmentesítés: A kép beolvasása során könnyen kerülhet zaj (kis pontok) a képbe különböző okok miatt, mint például a kamera nem megfelelő tisztasága, árnyékok stb. miatt. Ezeket a zajokat el kell távolítani, hogy a kép tiszta és egységes legyen.

- Vékonyítás és szkeletonizáció: Különböző képek szövege különböző vastagságú vonalakkal rendelkezik. A szkeletonizáció egy olyan technika, amely segítségével azonos szélességűvé tehetjük az összes vonalat.

1. ábra Vékonyítás és szkeletonizáció [4]

Szegmentálás: Ez egy olyan technika, amely részekre bontja a teljes képeket a további feldolgozáshoz. Háromféle szegmentálás végezhető el:

- Vonalszintű
- Szószintű
- Karakter szintű

Jellemzők kinyerése (Feature Extracion): A szegmentált alkomponensek néhány egyedi jellemzőjét vonjuk ki. Manapság a jellemzők kinyeréséhez leginkább gépi tanulási modelleket használunk, melyek CNN (Convolutional Neural Network), RNN (Recurrent Neural Network), LSTM (Long Short-Term Memory) rétegeket használnak.

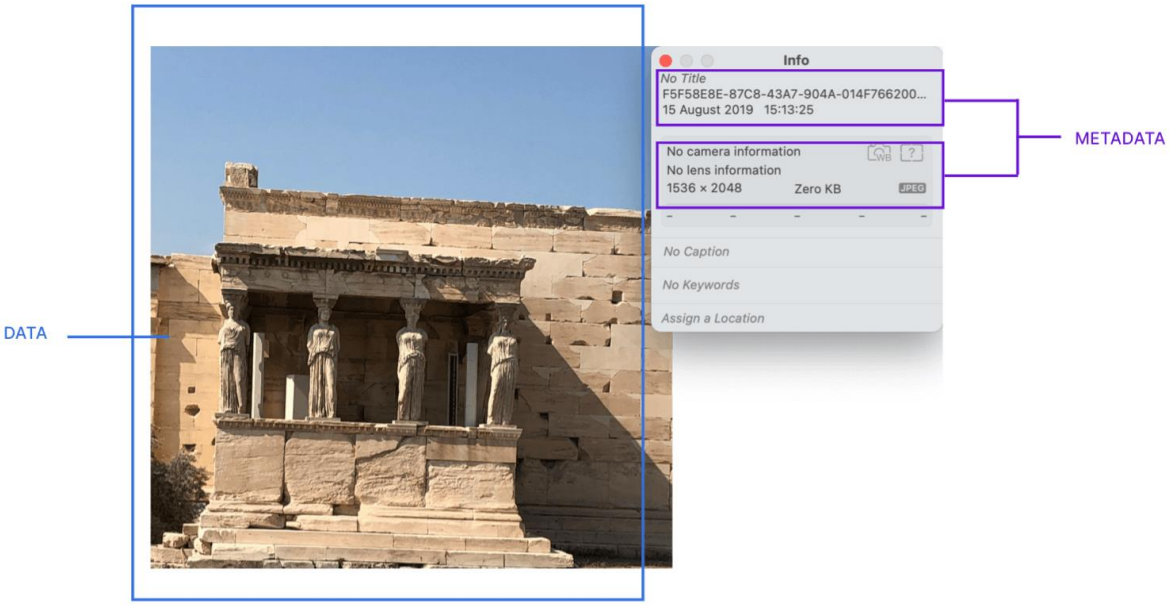
Klasszifikáció: Döntéshozatali szakasz. A szövegszegmensek azonosításához a kinyert jellemzőket használja. Ezen célra olyan algoritmusok használhatók, mint az SVM (Support Vector Macine).

Utófeldolgozás: A legvalószínűbb hiba a rossz predikció miatt következik be a klasszifikációs szakaszban. Ennek az oka lehet a képen lévő sok zaj, rossz jellemző kinyerés stb. Az esetek többségében ezek a predikciós hibák kisebb helyesírási hibákat eredményeznek. Ezek a hibák nyelvi modellek, Word2Vec modellekkel korrigálhatók. [1] [2] [3] [4]

2.2 Metaadatok kezelése

2.2.1 A metaadat és típusai

A metaadatok olyan információkat tartalmaznak az adatokról, amelyek segíthetnek azok jellemzésében és rendezésében. Ezek lehetnek például egy fájl neve, mérete, rögzítési időpontja és helye. A metaadatok hasznosak lehetnek az adatok ellenőrzésében és nyomon követésében, valamint a rendszerezésükhöz. Számos szoftver használja ezeket az információkat az adatok értelmezéséhez és kategorizálásához, például a keresőmotorok a weboldalak csoportosításához. [5]



2. ábra Metaadat[8]

A metaadatokat különböző kategóriákba soroljuk az információkezelésben betöltött funkciójuk alapján.

- Adminisztratív metaadatok: Ezek az adatok segítik a rendszergazdákat az adatok hozzáférésének és felhasználói engedélyeinek kezelésében, valamint adatforrások karbantartására és kezelésére vonatkozó információkat is tartalmaznak.

- Leíró metaadatok: Az adatok konkrét jellemzőit azonosítják, mint például bibliográfiai adatok, kulcsszavak, dalcímek.

- Jogi metaadatok: Ide tartoznak a kreatív licenceléssel kapcsolatos információk, például a szerzői jogokra, licencelésre és jogdíjakra vonatkozó adatok.

- Megőrzési metaadatok: Ezek az adatok irányítják egy adatelem hierarchikus vagy sorrendi elrendezését.

- Folyamat metaadatok: Az eljárásokat vázolják fel, amelyeket statisztikai adatok gyűjtésére és kezelésére használnak.

- Származási metaadatok: Követik egy adat történetét, ahogy az egy szervezeten belül mozog, és biztosítják az adatok érvényességét vagy minőségét.

- Referencia metaadatok: A statisztikai tartalom minőségéről tartalmaznak információkat.

- Statisztikai metaadatok: A felhasználók számára a jelentésekben, felmérésekben és összeállításokban található statisztikák megfelelő értelmezését és felhasználását teszik lehetővé.

- Strukturális metaadatok: Összetett adatobjektumok elemeinek szerkezetét mutatják be, például digitális médiatartalmak szervezését.

- Használati metaadat: Olyan adatok, amelyek minden felhasználói hozzáférés alkalmával rendeződnek és elemződnek. Ezen adatok analízise után az üzleti vállalkozások könnyebben igazíthatják termékeiket és szolgáltatásaikat a felhasználói igényekhez.
  [6] [7] [8]

2.2.2 A metaadatok kezelése

Az adatforrások elterjedése növeli a szervezeti adatok összetettségét. A metaadat-kezelés segít az adatok hatékony rendezésében és átlátható nyilvántartás létrehozásában, biztosítva az adatvédelmi előírásoknak való megfelelést. Általában olyan eszközökkel kezelik a metaadatokat, amelyek automatikusan rögzítik és tárolják azokat, például vállalati alkalmazásokban vagy adattárházakban. Azonban az AI és gépi tanulás által támogatott aktív metaadat-kezelők automatizálják a profilozást és címkézést, valamint segítenek a hibás vagy hiányzó adatok azonosításában. Ilyen népszerű eszköz például az Oracle Enterprise Metadata Management és az SAP PowerDesigner. A metaadat-kezelésnek számos előnye van, például az adatok egységesítése, amely segít az adatforrások közötti konzisztencia biztosításában és az adatértelmezési hibák csökkentésében. Emellett az automatizált metaadat-kezelő eszközök megelőzik a redundáns metaadatok felhalmozódását, így csökkentve a felesleges adattárolási költségeket. Továbbá javítja a termelékenységet, egyszerűsítve az adatok visszakeresését és csökkentve a szükséges időt a projektvezetésben, különösen nagy volumenű kutatási projektek esetében. [9]

2.3 NLP

2.3.1 Az NLP alapjai

A természetes nyelvfeldolgozás (Natural Language Processing - NLP) a mesterséges intelligenciát, a számítástechnikát és a nyelvészetet kombinálja, ezáltal lehetővé téve a számítógépek és digitális eszközök számára az emberi kommunikáció (például szöveg és beszéd) megértését és generálását. Az NLP teszi lehetővé számos olyan program működését, melyek képesek szövegeket fordítani, hosszabb szövegeket összefoglalni, igény szerint szöveg létrehozására, reagálnak begépelt vagy hang alapú utasításokra, sokszor valós időben. Felhasználása egyre elterjedtebb az üzleti környezetben is. Segít hatékonyabbá tenni a vállalati folyamatokat, automatizálni azokat, emelni a dolgozók hatékonyságát és egyszerűsíteni a kulcsfontosságú üzleti tevékenységeket.
Az emberi nyelv sokszor többértelmű is lehet, ami megnehezíti az olyan szoftverek fejlesztését, amelyek pontosan megértik a szöveg vagy hangfelvételek jelentését. Homonimák, homofonok, szarkazmus, idiómák, metaforák, nyelvtani különbségek és az eltérő mondatszerkezetek is mind kihívást jelentenek ebben a folyamatban. Több NLP-feladat azt célozza, hogy az emberi szöveges és hang adatokat úgy értelmezze, hogy a számítógép könnyebben megérthesse ezek tartalmát. Néhány ilyen feladat a következő: [10] [11]

- Beszédfelismerés: A hangadatok átalakítása szöveges formába. Szükséges minden olyan alkalmazásban, ami hangutasításokat követ, vagy szóban elhangzó kérdésekre válaszol. Az emberek beszéde kihívást jelent ebben, mivel gyors, gyakran összemosva a szavakat, különböző hangsúlyokkal és akcentusokkal, valamint gyakran helytelen nyelvtani kifejezéseket használnak.

- Beszédrészjelölés: A szó vagy kifejezés nyelvtani jelölése az adott kontextusban történő meghatározása annak alapján, hogy hogyan van használva a szó vagy kifejezés a szövegben. Például a „make” szó lehet igeként használva az „I can make a paper plane” (Tudok papírrepülőt készíteni) mondatban, és főnévként a „What make of car do you own?” (Milyen márkájú autója van?) mondatban.

- Szóértelmezési diszambiguació: Feladata a többértelmű szavak megfelelő jelentésének kiválasztása. Ez a folyamat a kontextus alapján azonosítja a legmegfelelőbb jelentést. Például segít megkülönböztetni a "make" ige "elérni" (make the grade) és "elhelyezni" (make a bet) jelentéseit.

- Nevesített entitások felismerése: Az entitások nevesített azonosítása során olyan szavakat vagy kifejezéseket azonosítunk, amelyek fontos entitásokra utalnak. Például azonosítjuk "Budapestet"-t mint egy helyet vagy "Fred"-et mint egy férfi nevet.

- Társreferenciák felbontása: Annak azonosítása, hogy két szó vagy kifejezés ugyanarra az entitásra utal-e, és ha igen, mikor. Például meghatározzuk, hogy egy adott névmás melyik személyt vagy tárgyat jelöli (például „ő” = „Mária”), de ez lehetőséget ad a metaforák vagy idiómák azonosítására is (például, ha a „medve” egy nagy termetű, szőrös embert jelöl).

- Természetes nyelvi generálás: Néha a beszédfelismerés vagy a beszédből szöveggé alakítás ellentéteként írják le. Ekkor a cél az emberi nyelvbe ágyazott strukturált információk megteremtése.

- Érzelemvizsgálat: Az NLP-technika áttekinti a szöveget annak érdekében, hogy felismerje benne az érzelmeket (például a „pozitív”, „negatív” vagy „semleges”). A vállalkozások gyakran alkalmazzák az érzelemelemzést annak érdekében, hogy jobban megértsék a vásárlói visszajelzéseket.

- Összegzés: Összegezni lehet hosszabb szövegeket, hogy gyorsabban átláthatóvá váljanak az időszűkében lévő olvasók számára. Ezek az összefoglalások érintik a jelentéseket és a cikkeket is.

- Kulcsszó kinyerés: Lényege az alapvető kifejezések és kulcsszavak felismerése, ami fontos szerepet játszik keresőmotor-optimalizálásban, közösségi médiafigyelésben és üzleti intelligencia területén.

- Tokenizálás: A karakterek, szavak vagy részszavak szétválasztása "tokenekre", amelyeket egy szoftver további elemzésre tud használni. A tokenizálás gyakran alkalmazott eljárás a természetes nyelvfeldolgozásban, például szómodellezés, szókincsépítés és gyakori szavak előfordulásának elemzése terén. [10] [11]

2.3.2 Az NLP előnyei és felhasználási területei

Az NLP számos előnyét kihasználva vállalkozások képesek strukturált és strukturálatlan adatok elemzésére, beleértve a beszédeket, szöveges üzeneteket és közösségi médiás bejegyzéseket. Ezáltal javíthatják az ügyfélelégedettséget és -élményt, miközben csökkentik a költségeket az NLP-alapú mesterséges intelligencia alkalmazásával, például chatbotok segítségével vagy nagy mennyiségű szöveges adat elemzésével. Ezen kívül az NLP segítségével jobban megérthetik célpiacukat és márkájukat azáltal, hogy releváns adatokat dolgoznak fel, mint például közösségi média bejegyzéseket vagy fókuszcsoportos felméréseket.

Spam-felismerés, gépi fordítás, virtuális ügynökök és chatbotok, közösségi média hangulatelemzés és szövegösszefoglalás mind olyan területek, ahol az NLP technológiák nagy szerepet játszanak. Például a spam-felismerésnél az NLP segítségével az e-mailekben található gyanús tartalmakat azonosítják, míg a gépi fordításban az NLP technológiák pontos és értelmes fordításokat biztosítanak. A virtuális ügynökök és chatbotok beszédfelismerésre és természetes nyelvi generálásra használják az NLP-t, így hatékonyan tudnak válaszolni a felhasználók kérdéseire. A közösségi média hangulatelemzése során az NLP lehetővé teszi a közösségi média platformokon megjelenő érzelmek és attitűdök elemzését. Végül, a szövegösszefoglalásnál az NLP technológiák segítségével rövid és tömör összefoglalókat készítenek hosszabb szövegekből. [10] [11]

2.4 Generatív AI modellek

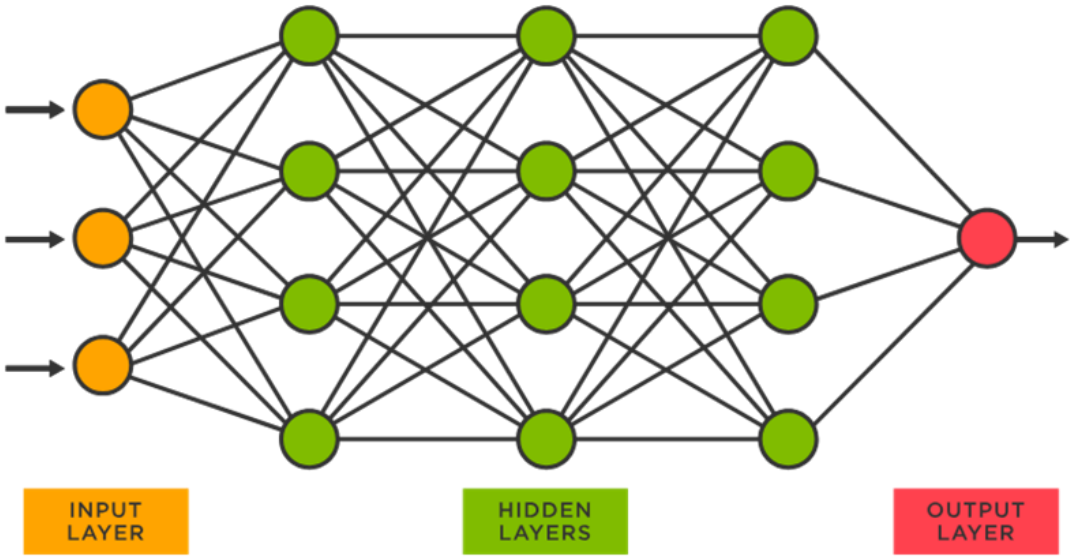
2.4.1 Neurális hálózatok és mélytanulás

A neurális hálózatok az emberi agy felépítéséhez hasonló információfeldolgozó rendszerek. A neurális hálózatok kisebb számítási egységekből, neuronokból állnak, amelyek összekapcsolódva képesek tanulni és komplex feladatokat megoldani. Minden kapcsolat rendelkezik egy súllyal (weight) ami meghatározza a kapcsolat erősségét és előjelét. [12] Alapvetően 3 rétegből állnak:

- Bemeneti réteg (Input Layer): Ez fogadja az adatokat, nem végeznek aktivációs függvényt. A bemeneti réteg neuronjainak száma általában megegyezik az adatok dimenziójával.

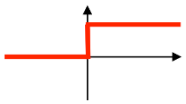
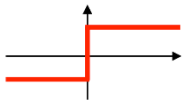
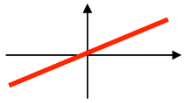
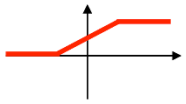
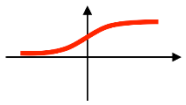
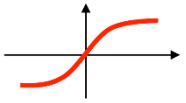
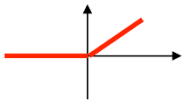
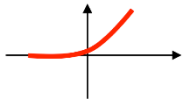
- Rejtett réteg (Hidden Layer): A bemeneti adatokon valamilyen átalakítást végez. Egy neurális hálózatban több rejtett réteg is lehet, a rejtett réteg neuronjainak száma és aktivációs függvénye a neurális hálózat tanulási képességét befolyásolja.

  Kimeneti réteg (Output Layer): A bemeneti adatokra adott választ adja vissza. A kimeneti réteg neuronjainak száma általában megegyezik a kívánt kimenet dimenziójával.



3. ábra Neurális hálózat általános ábra [14]

A tanulási folyamat adatokon keresztül történik, ahol a hálózatot előre definiált adatkészlettel tanítják, amely tartalmazza a helyes válaszokat is. A cél a súlyok beállítása, hogy a kimeneti rétegben a neuronok értéke a lehető legközelebb álljon az elvárt értékhez.
Az aktivációs függvények alapvető szerepet töltenek be a neurális hálózatok tanulási folyamatában. Ezek a függvények lehetővé teszik olyan kimeneti értékek előállítását a rejtett réteg neuronjainak, amelyek a bemeneti paramétereikben nem lineárisak. [13] [14]

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

4. ábra Fontosabb aktivációs függvények[5]

2.4.2 Generatív AI alapjai és típusai

A generatív mesterséges intelligencia olyan technológia, amely segítségével az emberek könnyedén létrehozhatnak új tartalmakat különböző bemeneti adatok alapján. Ezek a modellek képesek feldolgozni és megérteni a szöveges, képi, hangos, animált, 3D modellekhez hasonló vagy más típusú adatokat, majd válaszként létrehozni valami újat. [15] [16]

A generatív AI modellek neurális hálózatokat alkalmaznak a meglévő minták és struktúrák azonosítására, hogy új tartalmat hozzanak létre. Ezek az úttörő generatív

[5] Forrás: https://sebastianraschka.com/images/faq/activation-functions/activation-functions.png

mesterséges intelligencia modellek különböző tanulási módszereket használnak a képzés során, beleértve a felügyelet nélküli és félig felügyelt tanulást is. Ezáltal könnyedén használhatók nagy mennyiségű címkézetlen adatból alapmodellek létrehozásához, amelyek sokféle feladatot elláthatnak. Többnyire három fázisban működik:

- Tanítás (Training)
- Hangolás (Tuning)
- Generálás, értékelés, további hangolás (Generation, evaluation, more tuning) [16]

Generatív AI alapmodelljeinek készítése mélytanulási algoritmusokkal kezdődik, melyek nagy adatmennyiséget tanítanak be anélkül, hogy előre címkézett lenne. Ezek a modellek különböző tartalmak létrehozására alkalmasak, mint például szövegek, képek, hangok vagy zenék generálására. A tanítás során az algoritmusok próbálják megjósolni a következő elemet a sorozatokban úgy, hogy jóslatok és a valóság közötti különbséget minimalizálják. Ez az eljárás nagy számítási erőforrásokat és időt igényel, de nyílt forráskódú projektek segítségével az AI-fejlesztők képesek kihagyni ezt a lépést és csökkenteni a költségeket. [16]

Az alapmodell, bár sokféle tartalommal képes dolgozni, specifikus feladatokhoz hangolást igényel. Ez a folyamat két fő módon valósulhat meg. Az első módszer a finomhangolás, amely során a modellt az adott alkalmazáshoz vagy feladathoz jellemző címkézett adatokkal kell táplálni. Például, ha egy fejlesztőcsapat egy ügyfélszolgálati chatbotot próbál létrehozni, akkor a modellhez számos dokumentumot adnak, amelyek címkézett ügyfélszolgálati kérdéseket és a megfelelő válaszokat tartalmazzák. Ezáltal a modell a specifikus feladatokhoz könnyebben alkalmazkodik. A finomhangolás azonban munkaigényes folyamat. Gyakran a fejlesztők kiszervezik ezt a feladatot olyan vállalatoknak, amelyek nagy adatcímkéző munkaerővel rendelkeznek, hogy hatékonyan végezhessék el azt. A második módszer az erősítéses tanulás emberi visszajelzéssel (RLHF), amely során az emberi felhasználók értékelik a generált tartalmat. Az ilyen értékelések segítik a modell frissítését a nagyobb pontosság vagy relevancia érdekében. Az emberek gyakran különböző kimeneteket értékelnek, vagy egyszerűen csak javítják a chatbot vagy virtuális asszisztens generált válaszait, így finomítva a modellek teljesítményét. Ezáltal az RLHF fontos eszköz a modell finomhangolásában és fejlesztésében. [16]

A fejlesztők és felhasználók rendszeresen kiértékelik a generatív AI alkalmazások eredményeit, és finomítják a modellt nagyobb pontosság vagy relevancia érdekében, akár hetente is. Az alapmodellt viszont ritkábban frissítik, esetleg évente vagy 18 hónaponként. Egy másik lehetőség a generatív AI alkalmazás teljesítményének javítására a visszakeresésen alapuló kiterjesztett generálás (RAG). Ez a keretrendszer kiegészíti az alapmodellt olyan releváns forrásokkal, amelyek nem szerepelnek a

képzési adatok között, így finomítja és kiegészíti az eredeti modell paramétereit vagy reprezentációit. A RAG biztosítja, hogy a generatív AI alkalmazás mindig friss információkhoz férjen hozzá. Ráadásul a RAG-on keresztül elérhető további források átlátható módon megjelennek a felhasználók számára, ellentétben az eredeti modellbeli tudással. [16]

A generatív mesterséges intelligencia modellek különböző fajtái léteznek, melyek mindegyike egyedi módon közelíti meg a tartalom előállítását. Néhány generatív modell típus:

- Generatív adverzális hálózatok (GAN): A GAN-ok két neurális hálózatot használnak: egy generátort, amely szintetikus adatokat (pl. képeket, szöveget, hangot) hoz létre, és egy diszkriminátort, amely megkülönbözteti a valódi adatokat a hamisaktól. A generátor egyre hitelesebb adatokat készít, míg a diszkriminátor folyamatosan javítja az elkülönítési képességét. Ez a versengés lehetővé teszi a GAN-ok számára, hogy nagyon élethű tartalmakat generáljanak, amelyeket képszintézisben, művészetben és videókészítésben használnak.

- Variációs automatikus kódolók (VAE): A VAE-k generatív modellek, amelyek adatokat kódolnak egy látens térbe, majd visszafejtik azokat az eredeti adatok rekonstruálásához. Ezek a modellek valószínűségi reprezentációkat tanulnak, és új mintákat hoznak létre a megtanult eloszlás alapján. Gyakran alkalmazzák kép-, szöveg- és hanggenerálásra.

- Autoregresszív modellek: Az autoregresszív modellek olyan algoritmusok, amelyek új elemeket hoznak létre, miközben azok a korábban generált elemektől függenek. Például a GPT ((Generative Pre-trained Transformer)) nyelvi modell képes összefüggő szöveget előállítani azzal a tudással, amit az előző részekből merített. Tehát az autoregresszív modellek a kontextus alapján generálnak új adatokat.

- Rekurrens neurális hálózatok (RNN): Az RNN-ek olyan típusú neurális hálózatok, melyek szekvenciális adatokkal dolgoznak, mint például természetes nyelvi mondatok vagy idősorok. Feladatuk generatív, előrejelzést tesznek az előző elemek alapján a következő elemre. Ugyanakkor hosszú szekvenciák esetén problémába ütközhetnek az eltűnő gradiens miatt. A korlátok leküzdésére fejlesztették ki az LSTM-et (Long Short-Term Memory) és a GRU-t (Gated Recurrent Unit).

- Transzformátor-alapú modellek: A GPT és hasonló transzformátorok népszerűek a természetes nyelvi feldolgozásban és generatív feladatokban. Figyelemmechanizmusokat alkalmaznak a szekvenciák közötti kapcsolatok

hatékony modellezésére. Ezek a modellek párhuzamosíthatók és képesek kezelni hosszabb szövegeket is, így ideálisak összefüggő és kontextusban releváns tartalmak generálására.

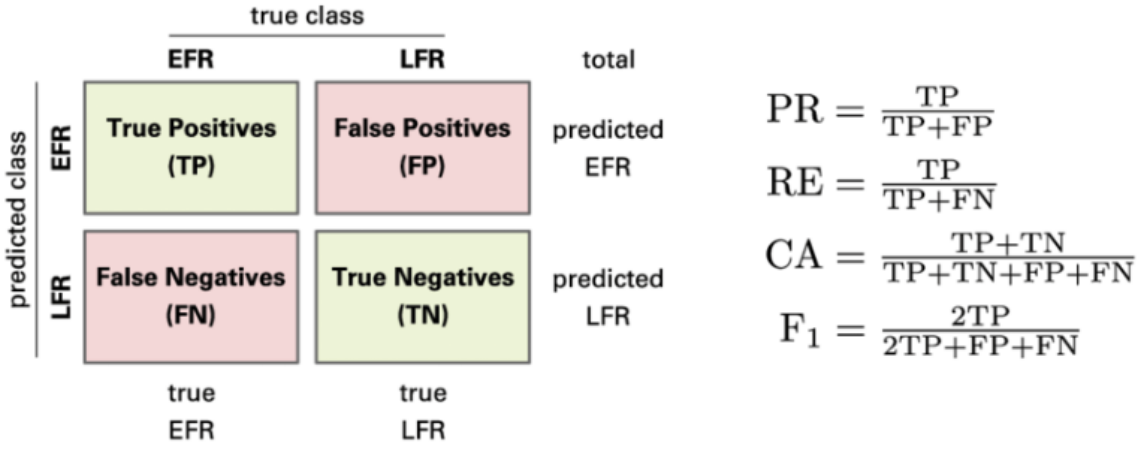
- Megerősítéses tanulás generatív feladatokhoz: A megerősítéses tanulás alkalmazható generatív feladatokra is. Ebben a megközelítésben egy ágens interakcióba lép a környezettel, és visszajelzést kap a generált adatok minősége alapján. Ezt a módszert például szöveggenerálásban használják, ahol a megerősítéses tanulás segíti a generált szövegek finomhangolását a felhasználói visszajelzések alapján.
  [15] [16] [17]

2.5 Értékelési metrikák

2.5.1 Fontosabb értékelési metrikák

Konfúziós mátrix (Confusion Matrix):

A paraméterek egy mátrix formájában jelennek meg egy konfúziós mátrix segítségével, ami lehetővé teszi a valódi és hamis pozitív, valamint a valódi és hamis negatív eredmények megtekintését. Az általános pontosság kiszámításához eloszthatjuk a tesztek teljes számát a hamis pozitív és hamis negatív eredmények számával. [18] [19]



5. ábra Konfúziós mátrix, Precizitás, Visszahívás, Pontosság, F1 pontszám képletek [18]

Precizitás (Precision):

Azt mutatja meg, hogy a helyesen előrejelzett esetek hány százaléka volt valóban pozitív. A precizitás különösen akkor hasznos, ha a Hamis Pozitív eredmények nagyobb problémát jelentenek, mint a Hamis Negatívak.

A precizitását úgy határozzuk meg, hogy a valódi pozitívumok számát elosztjuk a megjósolt pozitívumok számával. [18] [19]

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

6. ábra Precizitás képlete [19]

Visszahívás (Recall):

Megmutatja, hogy a valós pozitív esetek közül mennyit sikerült helyesen azonosítania a modellünknek. A visszahívás olyan helyzetekben hasznos mérőszám, amikor a hamis negatív eredmények komolyabb problémát jelentenek, mint a hamis pozitívak.

A visszahívását úgy határozzuk meg, hogy a valódi pozitív esetek számát elosztjuk a tényleges pozitív esetek teljes számával. [18] [19]

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

7. ábra Visszahívás képlete [19]

Pontosság (Accuracy):

A pontosság azt mutatja meg, hogy az osztályozó milyen gyakran ad helyes előrejelzés.

A pontosság kiszámítása a helyes előrejelzések számának elosztásával történik az összes előrejelzés számával. [18] [19]

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

8. ábra Pontosság képlete [19]

F1 pontszám (F1 Score):

A pontosság és a visszahívás harmonikus átlaga. Magasabb pontszámot kapunk, ha a modell mindkét mutatóban jól teljesít. Ha bármelyik mutató alacsony, az F1 pontszám is jelentősen csökken. A magas F1 pontszámú modelleknél az "igaz:hamis" pozitív és negatív arányok szélsőségesek, például 100:1 arányú valódi és hamis pozitív eredmények.

Ezzel szemben, ha az arányok közel vannak egymáshoz, mint 50:51, az F1 pontszám alacsony lesz.

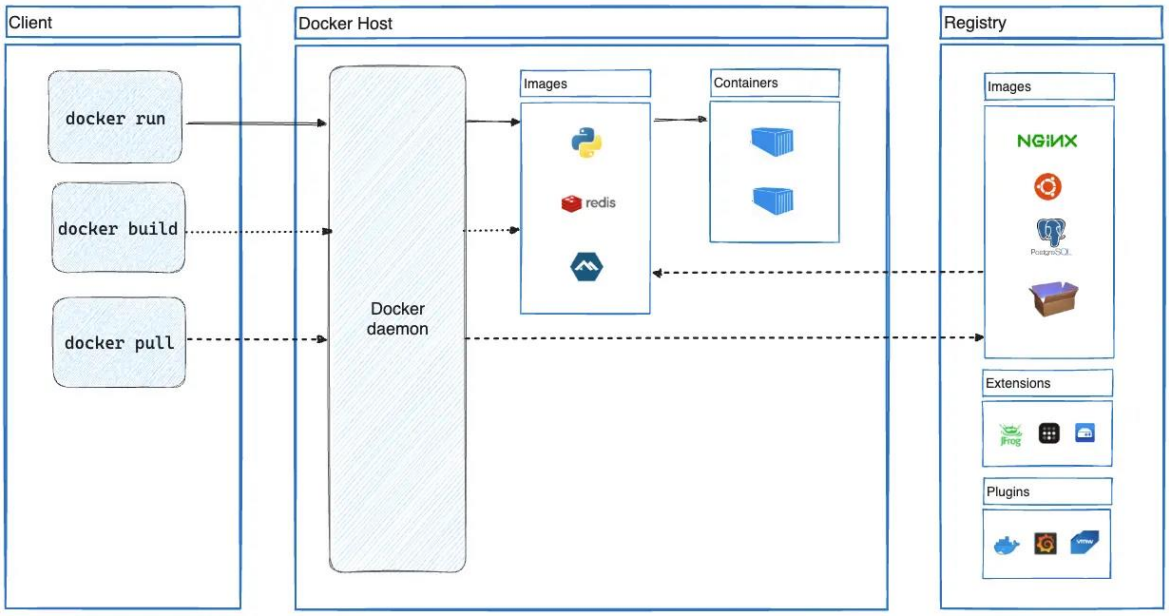
$$F1 = \frac{2 \times Precision \times Recall}{Precision + Recall}$$

9. ábra F1 pontszám képlete

2.6 Docker konténerizáció

2.6.1 Docker alapjai és előnyei

A Docker egy nyílt konténerizációs platform, amely lehetővé teszi szoftverek csomagolását és futtatását konténerekben. Ezek a konténerek elkülönített környezetek, amelyek magukban foglalják az összes szükséges függőséget és konfigurációt, így a szoftver környezettől függetlenül futtatható. A konténerek erőforrás-hatékonysága szintén kiemelkedő, mivel kevesebb erőforrást igényelnek, mint a hagyományos virtuális gépek. A Docker kliens-szerver architektúrát használ. A Docker-kliens a Docker-démonnal kommunikál, amely a konténerek létrehozását és futtatását végzi. A Docker-kliens lehet egy rendszeren a Docker-démonnal, vagy a kliens csatlakoztatható egy távoli Docker-démonhoz. A Docker-kliens és a démon REST API segítségével, UNIX socketeken vagy hálózati interfészen keresztül kommunikál. Egy másik Docker-kliens a Docker Compose, amely lehetővé teszi, hogy konténerekből álló alkalmazásokkal dolgozzunk. [20] [21]



10. ábra Docker kliens-szerver architektúra [20]

A Docker konténerizáció előnyei:

- Hordozhatóság: A konténerekbe csomagolt alkalmazásokat bármilyen környezetben futtathatjuk, legyen szó virtuális vagy fizikai szerverről, helyi adatközpontról vagy nyilvános felhőről. Nem kell aggódnunk az alkalmazás függőségeinek telepítése vagy konfigurálása miatt, mivel azok a konténerben vannak.

- Hatékonyság: A konténerek kevesebb erőforrást igényelnek, mint a hagyományos virtuális gépek, mivel nem kell emulálni egy teljes operációs rendszert és hardvert. A konténerek gyorsabban indulnak és leállnak, mint a virtuális gépek, és több alkalmazást tudunk futtatni egyetlen szerveren.

- Biztonság: A konténerek izolálják az alkalmazásokat egymástól és a gazda rendszertől, így csökkentik a potenciális támadási felületet. A konténerek könnyen frissíthetők és visszaállíthatók, így a sérülékenységek gyorsan javíthatók vagy elkerülhetők.

- Modularitás: A konténerek lehetővé teszik az alkalmazások felbontását kisebb, független és újrahasznosítható egységekre, amelyek egymással kommunikálhatnak. Ez növeli az alkalmazások rugalmasságát, skálázhatóságát és karbantarthatóságát.
  [22]

2.6.2 Konténerizáció szerepe a fejlesztési folyamatban

A konténerizáció szerepe a fejlesztési folyamatban, különösen a Docker esetében, forradalmasította a szoftverfejlesztést. A Docker egységesíti a fejlesztési és üzemeltetési környezeteket, megkönnyítve az átmenetet a fejlesztéstől az éles környezetig. A Docker tökéletesen integrálódik a CI/CD (Folyamatos Integráció/Folyamatos Telepítés) folyamatokkal, lehetővé téve az automatizált tesztelést és kiadást. A mikroszolgáltatásokra alapozott architektúrák hatékony implementálásában is kulcsfontosságú, ahol minden szolgáltatás külön konténerben fut.
A konténerek konzisztenciát biztosítanak, amely lehetővé teszi, hogy az alkalmazások ugyanúgy viselkedjenek a fejlesztési, tesztelési és éles környezetekben. Ezáltal a környezetfüggő hibák és inkompatibilitások elkerülhetők. Továbbá a konténerek egységesítik a fejlesztési és üzemeltetési eszközöket és platformokat, amivel csökkentik a komplexitást és a függőségeket. A konténerek gyorsaságot is hoznak az alkalmazások létrehozásának, telepítésének és frissítésének folyamatába. Mivel nem kell várni a virtuális gépek indítására vagy a függőségek telepítésére, a fejlesztők gyakrabban és kisebb lépésekben adhatják ki az alkalmazás változatait, ezzel növelve a termelékenységet és a minőséget. A kollaboráció szintén javul a konténerek használatával. Megkönnyítik az alkalmazások megosztását és együttműködést a fejlesztők, tesztelők és üzemeltetők között. A konténerek egyszerűen másolhatók, átvihetők és futtathatók, támogatva a mikroszolgáltatások architektúráját, amely lehetővé teszi a fejlesztők számára, hogy különálló, független és egyszerűbb modulokon dolgozzanak. Végül, a Docker konténerek skálázhatósága kiemelkedő. Az automatizált orchesztrációs eszközök, mint a Kubernetes, képesek kezelni a Docker konténereket, automatikusan skálázva őket a terhelésnek megfelelően. Ezek az előnyök összességében jelentős előrelépést jelentenek a szoftverfejlesztés terén, és hatalmas hatással vannak a fejlesztési folyamatokra és az üzemeltetési hatékonyságra. [23] [24]

2.7 Konklúzió

A generatív mesterséges intelligencia és a természetes nyelvfeldolgozás alkalmazása az optikai karakterfelismerés (OCR) technológiával nyert szövegek kezelésére és javítására ígéretes megoldásnak bizonyult. A dolgozatban bemutatott technikák és módszerek segítségével jelentősen javítható a digitalizált szövegek pontossága és minősége, amely különösen fontos a nyomtatott és kézzel írott dokumentumok esetében. Az OCR technológia által kinyert szövegek gyakran tartalmaznak hibákat, amelyek javítása és feldolgozása komoly kihívást jelent. A generatív AI és NLP technikák azonban hatékony eszközöket kínálnak ezen problémák megoldására.

A Docker konténerizáció alkalmazása lehetővé teszi a fejlesztési folyamatok hatékonyabbá tételét, biztosítva a szoftverek hordozhatóságát és hatékony futtatását különböző környezetekben. Az automatizált és skálázható megoldások révén a generatív AI és az NLP technológiák könnyen integrálhatók a meglévő rendszerekbe, jelentősen javítva az OCR-ezett szövegek feldolgozásának hatékonyságát.

A dolgozatban bemutatott megközelítések és technológiák összességében hozzájárulnak a digitalizált szövegek pontosságának és használhatóságának növeléséhez. Az adatok előfeldolgozása, normalizálása és tisztítása alapvető lépések a modellek hatékonyságának növelése érdekében. A generatív AI és NLP technikák alkalmazása révén lehetővé válik az OCR technológia által kinyert szövegek pontosabb értelmezése és javítása, ezáltal növelve a digitalizált adatok értékét és hasznosságát.

Összegzésképpen, a generatív mesterséges intelligencia és a természetes nyelvfeldolgozás alkalmazása az OCR-ezett szövegek kezelésére és javítására jelentős előnyökkel járhat. Az AI technológiák kombinálása az OCR technológiával új lehetőségeket teremt a dokumentumkezelés és az adatfeldolgozás területén, biztosítva a digitalizált szövegek magasabb szintű pontosságát és használhatóságát. A Docker konténerizáció révén pedig a fejlesztési folyamatok hatékonysága és a szoftverek hordozhatósága tovább növelhető, támogatva ezzel a modern AI technológiák hatékony integrálását a meglévő rendszerekbe.

3. Rendszerterv

Ez a rendszer egy dokumentumfeldolgozó és címkekezelő eszközt valósít meg, amely különböző formátumú dokumentumokból automatikusan kinyeri a szöveget, és mesterséges intelligencia modellek segítségével címkéket generál hozzájuk. Ezeket a címkéket és az eredeti tartalmat később jegyzetként tárolja az Obsidianban, hogy hatékonyabb tudásrendezést és gyorsabb visszakeresést biztosítson.

3.1 A rendszer célja

A rendszer fő célja, hogy egy automatikus dokumentumfeldolgozó eszközt hozzon létre, amely:

- Könnyíti a dokumentumok áttekintését és címkézését.
- Támogatja a tudásmenedzsmentet az Obsidianban, biztosítva a címkék közötti kapcsolatok kezelését.
- Segít időt spórolni a manuális címkézés és rendezés kiváltásával.

3.2 Technológiák

Programozási Nyelv: Python

AI Modellek: OpenAI GPT és Huggingface Llama

Dokumentumkezelés: PyPDF2, python-docx, pytesseract

Szövegelemzés: SpaCy

Jegyzetkezelés: Obsidian alkalmazással történő integráció YAML metadata formátumban.

Környezeti Változók Kezelése: Egy .env fájl segítségével konfigurálható az OpenAI API kulcs, a Llama modell neve, a Spacy Modell neve és az Obsidian Vault elérési útja.

Verziókövetés: Git, GitHub

3.3 Rendszer architektúra

A rendszer architektúrája moduláris felépítésű, ezzel biztosítva a rugalmasságot és bővíthetőséget.

3.3.1 Fő modul

Az App modul felelős a program belépési pontjáért. Inicializálja az egész folyamatot, és a parancssori bemenetek alapján irányítja a működést.

3.3.2 Agent modulok

MainAgent: A fő folyamatot vezérli, meghívva a különböző funkciókat, mint a dokumentum betöltése, címkék generálása, előfeldolgozás és a jegyzetek mentése.

DocumentLoaderAgent: Felelős a különböző típusú dokumentumok szövegének kinyeréséért.

PreprocessingAgent: Az előfeldolgozást végzi, pl. írásjelek és számok eltávolítása.

LabelGeneratorok:

- GPTLabelGenerator: A GPT modellt használja címkék generálására.

- LlamaLabelGenerator: A Llama modellt használja alternatív címkegenerálásra.

ObsidianAgent: Az Obsidian vault kezeléséért felelős: jegyzetek mentése, címkék közötti kapcsolatok létrehozása.

3.4 Várható eredmények

A rendszer segítségével gyors és hatékony módon lehet dokumentumokat feldolgozni és címkézni, csökkentve a manuális munka igényét és biztosítva a jegyzetek egyszerű rendszerezését az Obsidian programban. A címkék közötti kapcsolatok felépítése javítja az információk kereshetőségét és rendszerezését, ami különösen hasznos lehet tudásmenedzsment rendszerekben.

4. Megvalósítás

4.1 Az alkalmazás futásához szükséges előkészületek

Az alkalmazás indításához parancssor szükséges. A parancssor megnyitását követően megkell keresni az alkalmazást tartalmazó mappát és abba belépni. A főkönyvtár megtalálását követően célszerű létrehozni egy virtuális környezetet tetszőleges névvel annak érdekében, hogy a program futásához szükséges könyvtárakat könnyebben kezelhessük.

```
1. PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> python -m venv myenv
```

A virtuális környezet létrehozását követően aktiválni is szükséges azt, melyet a következő parancs segítségével tehetünk meg.

```
1. PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> ./myenv/Scripts/activate
```

A virtuális környezet aktiválását követően megjelenik a jelenlegi mappa útvonala előtt egy, a virtuális környezet nevét tartalmazó jelzés.

```
1. (myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc>
```

Az aktiválás után érdemes mindenekelőtt futtatni egy pip frissítést az esetleges problémák, hibák elkerülése végett.

```
1. (myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> python -m pip install
--upgrade pip
```

Ezt követi a szükséges könyvtárak telepítése, melyeket a requirements.txt fájl tartalmazza.

```
1. PyPDF2==3.0.1
2. openai==1.54.4
3. python-dotenv==1.0.1
4. torch==2.5.1
5. transformers==4.46.2
6. spacy==3.8.2
7. accelerate===1.1.1
8. debugpy===1.8.8
9. python-docx===1.1.2
10. pytesseract===0.3.13
```

Ezen könyvtárak egyenkénti telepítése egy parancs segítségével elkerülhető, és lehetővé teszi az összes, a fájlban felsorolt könyvtár telepítését.

```
1. (myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> pip install -r requi-
rements
```

A könyvtárak telepítését követően szükséges még telepíteni a Tessract OCR (https://github.com/UB-Mannheim/tesseract/wiki) programot, mivel a pytesseract könyvtár ezt használja működésekor.

Mivel az alkalmazás a Spacy könyvtárat is használja generált címkék összehasonlítására, így szükséges egy megfelelő méretű és nyelvű modell telepítése. Az alkalmazás megvalósítása során három különböző méretű modell került tesztelésre. Ezek angol nyelvű modellek, de magyar tartalom feldolgozása esetén telepíthető magyar nyelvű modell is. Az modell méretét érdemes az alkalmazást futtató számítógép teljesítményéhez mérten megadni. Egy alacsonyabb teljesítményű számítógépen egy nagyobb modell lassabb működést eredményezhet. A fejlesztés során alkalmazott modell opciók fentről lefele haladva növekvő sorrendben az alábbiak.

```
1. python -m spacy download en_core_web_sm
2. python -m spacy download en_core_web_md
3. python -m spacy download en_core_web_lg
```

Az alkalmazás nagy nyelvi modelleket (LLM) használ. Ezek közül az egyik, az OpenAI által fejlesztett GPT-4o-mini, melyet OpenAI API kulccsal hívunk meg, a másik pedig a Meta által közzétett Llama-3.2-3B-Instruct. Ezek közül a Llama modell nagyobb mértékben konfigurálható. Ezen modellek elérése érdelében a .env fájl szerkesztése szükséges. Az egyes változók értének megkell adni az OpenAI API kulcsot, a HuggingFace Tokent, az Obsidian Vault elérési útvonalát, hogy egyenesen oda tudja menteni a generált címkéket, valamint szükséges megadni a telepítésre került Spacy modell nevét.

```
1. OPENAI_API_KEY=sk-proj-...
2. HUGGINGFACE_TOKEN=hf_...
3. LLAMA_MODEL_NAME=meta-llama/Llama-3.2-3B-Instruct
4. OBSIDIAN_VAULT_PATH=D:\Obsidian Vault\Thesis\Thesis_Vault
5. SPACY_MODEL=en_core_web_lg
```

Végül, a futás során előállított címkék és kapcsolatok reprezentálása, és könnyen áttekinthetősége végett telepítendő az Obsidian nevű program, mely a következő címen érhető el: https://obsidian.md/. A telepítést követően szükséges létrehozni egy ”Vault”-ot, amely elérési útvonalát szinten a korábban említett .env fájlon belül található megfelelő helyre szükséges megadni annak érdekében, hogy az alkalmazás oda tudja menteni a generált címkéket.

Amennyiben az alkalmazást futtató számítógép rendelkezik GPU-val, célszerű feltelepíteni további könyvtárakat és illesztő programokat annak érdekében, hogy a Llama modell azon futhasson, így gyorsabb működés érhető el.

```
1. pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
```

4.2 Az alkalmazás működése

4.2.1 Indításhoz szükséges parancs

A szükséges könyvtárak és illesztőprogramok telepítését, valamint a környezeti változók beállítását követően az alkalmazás indítása parancssorból történik. A program neve app.py, ezt kell beírni a parancssorba egy python utasítást követően. Ezután elindul a program, és a feldolgozás megkezdése előtt interakcióba lép a felhasználóval, mivel megkell adni a programnak a használni kívánt modell nevét. Válaszként csak a ”gpt” vagy a ”llama” érvényes. Miután ezt az adatot megkapja program, megkezdi a dokumentumok feldolgozását.

```
1. (myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> python app.py
2. Select model to use (gpt/llama): gpt
```

```
1. (myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> python app.py
2. Select model to use (gpt/llama): llama
```

4.2.2 app.py

A program az alap Python logging könyvtárat használja erre a célra, mely biztosítja, hogy minden fontos művelet, hiba vagy figyelmeztetés naplózásra kerüljön. A naplózási szint az INFO. Ez azt jelenti, hogy a program az információkat és az annál magasabb prioritású eseményeket fogja naplózni. További naplózási szint lehet még a: DEBUG, WARNING, ERROR. A naplózási formátum első helyén a bejegyzés időpontja szerepel, ezt követi naplózási szint, az aktuális metódus neve, és végül maga az üzenet zárja bejegyzést.

```
1. def setup_logging():
2.     logging.basicConfig(
3.         level=logging.INFO,
4.         format='[%(asctime)s] %(levelname)s - %(name)s: %(message)s',
5.         datefmt='%Y-%m-%d %H:%M:%S',
6.     )
```

A get_model_choice() metódus feladata, hogy a program indítását követően kérje a felhasználótól a használni kívánt modellt. Miután a felhasználó beírta a modell nevét, a metódus levágja az esetleges szóközöket a karakterlánc végéről, majd pedig minden karaktert kisbetűs formára alakít, ezzel csökkentve az esetleges hibák lehetőségét. A kapott választ megvizsgálja, hogy megtalálható egy listában melyben a két modell neve van, és ennek függvényében ad vissza igaz vagy hamis logikai értéket.

```
1. def get_model_choice():
2.     while True:
3.         choice = input("Select model to use (gpt/llama): ").strip().lower()
4.         if choice in ["gpt", "llama"]:
5.             return choice == "gpt"
6.         print("Invalid choice. Please enter 'gpt' or 'llama'.")
```

A feldolgozandó dokumentumokat tartalmazó mappa útvonalát a környezeti változókból nyeri ki a program, melyhez a get_folder_path_from_env() metódust használja. A metódus kinyeri a már beállított környezeti változókból az elérési útvonalat tartalmazót, majd ellenőrzi, hogy létezik-e a megadott mappa. Amennyiben létezik, visszaadja a hívó metódusnak az elérési útvonalat.

```
1. def get_folder_path_from_env():
2.     folder_path = os.getenv("DOCUMENT_FOLDER")
3.     if not folder_path or not os.path.isdir(folder_path):
4.         raise ValueError("Invalid or missing DOCUMENT_FOLDER in .env file.")
5.     return folder_path
```

Először a main kerül meghívásra. Ez a fő metódus, innen indul el az egész folyamat. A main-en belül először a környezeti változókat állítja be. Az override paraméter lehetővé teszi, hogy mindenképp beolvassa a .env fájlt és felülírja a már esetlegesen létező környezeti változókat. Ezt követi a naplózás beállítása, majd pedig a használni kívánt modell nevének bekérése a felhasználótól. Amennyiben a felhasználó érvényes modell nevet adott meg, beállítja a dokumentumokat tartalmazó mappa elérés útvonalát. Egy változóba kigyűjti a mappában található fájlok elérési útvonalát, majd ezeken végigiterálva, egyesével mindre meghívja a MainAgent osztály process metódusát, és feldolgozza a tartalmukat.

```
 1. def main():
 2.     load_dotenv(override=True)
 3.     setup_logging()
 4.     logger = logging.getLogger(__name__)
 5.
 6.     use_gpt = get_model_choice()
 7.
 8.     try:
 9.         folder_path = get_folder_path_from_env()
10.         files = [os.path.join(folder_path, filename) for filename in
os.listdir(folder_path) if os.path.isfile(os.path.join(folder_path, filename))]
11.         files.sort()  # Process files in sorted order
12.
13.         for file_path in files:
14.             logger.info(f"Processing file: {file_path}")
15.             agent = MainAgent(file_path, use_gpt)
16.             labels = agent.process()
17.
18.             if labels:
19.                 logger.info(f"Generated labels for
{os.path.basename(file_path)}:")
20.                 for label in labels:
21.                     print(f"- {label}")
22.             else:
23.                 logger.error(f"Failed to generate labels for
{os.path.basename(file_path)}.")
24.
25.     except Exception as e:
26.         logger.error(f"Fatal error occurred: {e}")
```

4.2.3 agent.py

Az MainAgent egy fő ágens szerepet tölt be, mely irányítja a szöveg betöltését, feldolgozását, a címkék generálását, a generált címkék kezelését és kimentését az Obsidian Vault-ba különböző metódusok és segéd ágensek segítségével. A MainAgent osztály inicializálása során először a konstruktor fut le, mely elvégzi az osztály példányosítását, az alapvető paraméterek és segédosztályok beállítását. A konstruktor kettő paramétert vár, ezek a beolvasni kívánt dokumentum elérési útvonala, valamint a használni kívánt LLM neve.

```
1. class MainAgent:
2.     def __init__(self, file_path, use_gpt):
3.         load_dotenv(override=True)
4.         self.logger = logging.getLogger(__name__)
5.         self.file_path = file_path
6.         self.use_gpt = use_gpt
7.         self.error_agent = ErrorHandlingAgent()
8.         self.obsidian_agent = ObsidianAgent()
9.         self.document_loader_agent = DocumentLoaderAgent(file_path)
10.        self.preprocessing_agent = PreprocessingAgent()
11.        self.label_generator = None
12.        self.nlp = spacy.load(os.getenv('SPACY_MODEL'))
```

A MainAgent osztály fő metódusa a process(), mely első lépésként beolvassa a dokumentumot a DocumentLoaderAgent osztály segítségével, majd pedig a PreprocessngAgent osztály segítéségével szövegtisztítást végez a jobb feldolgozhatóság elérésének érdekében. Miután a program sikeresen beolvasta és megtisztította a szöveget, beállítja a felhasználó által kiválasztott címke generálási modellt, majd meghívja a címkék generáláshoz szükséges GPTLabelGenerator, vagy LlamaLabelGenerator osztály generate_label() metódusát, mely elvégzi a releváns címkék generálását a tisztított szöveg alapján. Végül a generált címkéket egy változóban tárolja. A változóba mentett címkék egy karakter láncot alkotnak, és vesszővel vannak elválasztva, így ezeket feldaraboljuk és egy tömbben tároljuk. A feldarabolt címkéket ezután a program normalizálja, azaz egységes formára hozza annak érdekében, hogy összehasonlításuk és kezelésük egyszerűbb legyen a következő lépések során. Ezt a duplikáció elkerülése követ, melyet a program úgy kezel, hogy a már esetleg korábban, más beolvasott fájlhoz létrehozott címkéket, valamint az összes aktuálisan generált címkét összeveti egymással, és ha hasonló vagy azonos címkéket talál, gondoskodik róla, hogy csak egy ilyen címke jusson tovább ebben a folyamatban. Ezután a program az összes címkét megvizsgálva összeköti az egymással releváns címkéket, így kapcsolatot létrehozva közöttük. Végül pedig elmenti a generált címkéket az Obsidian Vault-ba.

```
1. def process(self):
2.         try:
3.             # Text scanning and pre-processing
4.             raw_text = self.document_loader_agent.document_to_text()
5.             clean_text = self.preprocessing_agent.clean_text(raw_text)
6.
7.             # Generate labels
```

```
 8.             all_labels = []
 9.             self.setup_label_generators()
10.             labels = self.label_generator.generate_labels(clean_text)
11.             self.logger.info(labels)
12.             if labels:
13.                 labels_list = [label.strip() for label in labels.split(',')]
14.                 all_labels.extend(labels_list)
15.             else:
16.                 self.logger.warning("Failed to generate labels with one of the
generators.")
17.
18.             if not all_labels:
19.                 self.logger.error("None of the generators could produce labels.")
20.                 return None
21.
22.             # Normalisation of labels
23.             normalized_labels = [self.normalize_label(label) for label in all_la-
bels]
24.
25.             # Scan existing labels
26.             existing_labels = self.obsidian_agent.get_existing_labels()
27.
28.             # Label matching with existing labels
29.             final_labels = []
30.             for label in normalized_labels:
31.                 similar_label = self.find_similar_label(label, existing_labels)
32.                 if similar_label:
33.                     final_labels.append(similar_label)
34.                 else:
35.                     final_labels.append(label)
36.                     existing_labels.add(label)
37.
38.             # Custom labels
39.             unique_labels = list(set(final_labels))
40.
41.             # Defining relationships between labels
42.             connections = self.connect_labels(unique_labels)
43.
44.             # Save note to Obsidian
45.             self.obsidian_agent.save_note_with_labels(
46.                 title=os.path.splitext(os.path.basename(self.file_path))[0],
47.                 content=raw_text,
48.                 labels=unique_labels
49.             )
50.
51.             # Create label notes with contacts
52.             self.obsidian_agent.create_label_notes(unique_labels, connections)
53.
54.             return unique_labels
55.         except Exception as e:
56.             self.error_agent.handle_error(e, "Processing error.")
57.             return None
```

A MainAgent konstruktor futásakor a használandó LLM beállításához egy külön metódust használ a program. A setup_label_generators() metódus megvizsgálja a korábban paraméterként megkapott logikai változót, mely során kiválasztása kerül a megfelelő LLM.

```
1. def setup_label_generators(self):
2.         try:
3.             if self.use_gpt:
```

```
                openai_api_key = os.getenv('OPENAI_API_KEY')
                gpt_generator = GPTLabelGenerator(api_key=openai_api_key)
                self.label_generator=gpt_generator
            else:
                llama_model_name = os.getenv('LLAMA_MODEL_NAME')
                llama_generator = LlamaLabelGenerator(model_name=llama_mo-
del_name)
                self.label_generator=llama_generator
        except Exception as e:
            self.error_agent.handle_error(e, "Label generator initialization er-
ror.")
```

A létrehozott címkék egységes formára alakításáért a normalize_label() metódus felel. Ez a folyamat lemmatizációval, valamint a kisbetűs formára alakítással biztosítja az egységes formát, ezzel csökkentve a címkék redundanciáját. Első lépésként minden karaktert kisbetűs formára alakít, ezt követően lemmatizálást hajt végre generált címkéken, mely biztosítja, hogy ne legyenek felesleges duplikációk.

```
def normalize_label(self, label):
        # Label normalisation: lower case and lemmatisation
        doc = self.nlp(label.lower())
        lemmatized = ' '.join([token.lemma_ for token in doc])
        normalized = lemmatized.strip()
        return normalized
```

A redundancia további csökkentéséért a find_similar_label() metódus is felel. Ezen folyamat célja, hogy az újonnan generált címkék esetében megkeresse azokat a már meglévő címkéket, amelyek hasonlóak/megegyeznek az új címkékkel. Ez a címkék egységesítésének és konszolidációjának alapvető lépése, amely segít a dokumentumok hatékonyabb rendszerezésében és kereshetőségében. A folyamat elején a paraméterként kapott címkét feldolgozza a Spacy modell segítségével, majd pedig végig iterál a már létező címkéken egyesével, és megvizsgálja a hasonlóságot. A hasonlóság küszöbértékének 0.8 lett beállítva. A küszöbérték 0 és 1 közötti értéknek kell lennie, ahol a 0 azt jelenti, hogy nincs hasonlóság, az 1 pedig, hogy pontosan megegyeznek. Ez alapján a 0.8-as érték azt mutatja, hogy pontos hasonlóságokat keres, minimális holtjátékkal.

```
def find_similar_label(self, label, existing_labels, threshold=0.8):
        # Search for similar labels among existing labels
        label_doc = self.nlp(label)
        for existing_label in existing_labels:
            existing_label_doc = self.nlp(existing_label)
            similarity = label_doc.similarity(existing_label_doc)
            if similarity >= threshold:
                return existing_label
        return None
```

A hasonlóság mérését egy másik metódusban is alkalmazza a program. A connect_labels() metódus a címkék közötti kapcsolatok feltárásáért és kialakításáért felel. A find_similar_label() metódushoz hasonlóan, Spacy modell-t használva feldolgozza a címkéket hogy könnyen összehasonlíthatók legyenek, ezt követően minden címke páros

hasonlóság vizsgálatot végez. A küszöbérték itt 0.7, ami minimálisan több rugalmasságot biztosít, ezáltal jobb kapcsolatokat biztosítva a címkék között. A kapcsolatokat egy tömbbe menti, amit végül visszaad a hívó metódusnak.

```
 1. def connect_labels(self, labels, threshold=0.7):
 2.         label_docs = {label: self.nlp(label) for label in labels}
 3.         connections = {}
 4.         for label1 in labels:
 5.             connections[label1] = set()
 6.             for label2 in labels:
 7.                 if label1 != label2:
 8.                     similarity = label_docs[label1].similarity(label_docs[la-
bel2])
 9.                     if similarity >= threshold:
10.                         connections[label1].add(label2)
11.         return connections
```

4.2.4 obsidian_agent.py

Az obsidian_agent.py az Obsidian jegyzetalkalmazással való integrációért felelős. Az ObsidianAgent osztály fő feladata, hogy a létrehozott jegyzeteket és címkéket elmentse az Obsidian Vault-ba, mely lehetővé teszi azok könnyebb elérését, áttekinthetőségét, ábrázolását, valamint a navigálást közöttük. Az ObsidianAgent osztály konstruktora kiveszi az Obsidian Vault elérési útvonalát környezeti változókat tartalmazó .env fájlból és beállítja azt, hogy később oda tudja menteni a generált címkéket, jegyzeteket.

```
 1. class ObsidianAgent:
 2.     def __init__(self):
 3.         load_dotenv(override=True)
 4.         self.logger = logging.getLogger(__name__)
 5.         self.vault_path = os.getenv('OBSIDIAN_VAULT_PATH')
 6.         if not self.vault_path:
 7.             self.logger.error("Obsidian vault path is not set.")
 8.             raise ValueError("Obsidian vault path is not set.")
 9.         if not os.path.exists(self.vault_path):
10.             self.logger.error(f"Obsidian vault path does not exist:
{self.vault_path}")
11.             raise FileNotFoundError(f"Obsidian vault path does not exist:
{self.vault_path}")
```

Az ObsidianAgent osztály get_existing_labels() metódusa felel a már létező címkék beolvasásáért, melyek az Obsidian Vaultban vannak tárolva Markdown fájlokként. A már létező címkéket a MainAgent osztály process() metódusa használja fel a duplikációk elkerülése érdekében. Első lépésként a metódus összefűzi az Obsidian Vault elérési útvonalát a Labels mappával, majd a Linux és Windows rendszerek közötti kompatibilitás miatt formázza a teljes elérési útvonalat. Ez azt jelenti hogy az esetlegesen előforduló ’\’ jeleket ’/’ jellel helyettesíti. Ezt követően ellenőrzi az elérési útvonal létezését. Amennyiben nem létezik, nem végzi el az ezt követő lépéseket. Amennyiben létezik, végig iterál a Labels mappában található címkéken, és hozzáadja egy tömbhöz az abban található

fájlok neveit normalizálva, kiterjesztés nélkül, mivel a címkéket tartalmazó fájlok nevei megegyeznek a címkékkel. Végül visszaadja a tömböt.

```
def get_existing_labels(self):
        labels = set()
        labels_dir = os.path.join(self.vault_path, 'Labels').replace('\\','/')
        if os.path.exists(labels_dir):
            for filename in os.listdir(labels_dir):
                if filename.endswith('.md'):
                    label = os.path.splitext(filename)[0]
                    labels.add(label.lower())
        return labels
```

A save_note_with_labels() metódus a MainAgent osztályban az Obsidian vaultba való jegyzetmentésért felel, és ennek során biztosítja, hogy a jegyzet tartalma, címe és hozzá tartozó címkék megfelelően kerüljenek tárolásra Markdown fájlok formájában. Először is, a metódus meghatározza a jegyzet fájlnevét a megadott cím alapján, és hozzárendeli a .md kiterjesztést, amely a Markdown formátumnak felel meg. Ezután a címkék listájából egy olyan szövegrész készül, amely kétféleképpen tartalmazza a címkéket: hashtag formában, ami az Obsidianban címkeként működik, és link formában, ami hivatkozásként szolgál a kapcsolódó címke-jegyzetre. Ezáltal a címkék egyszerre lesznek kereshetők és kapcsolódnak egymáshoz az Obsidianban, megkönnyítve a felhasználó számára a jegyzetek közötti navigációt, keresést. Ezt követően a metódus létrehozza a jegyzet YAML fejlécét, amely metaadatokat tartalmaz. Ez a rész a jegyzet címét, a létrehozás dátumát, valamint a címkék listáját foglalja magában. A YAML fejléc összeállítása után összeilleszti a teljes jegyzet tartalmát, melyet a YAML fejléc, a dokumentum szöveges tartalma, valamint a címkék alkotnak. Végül a metódus elmenti a jegyzetet Markdown fájlként az Obsidian Vault-ba a megadott elérési útvonalon.

```
def save_note_with_labels(self, title, content, labels):
        try:
            self.logger.info("Saving note to Obsidian...")
            # File name and path
            filename = f"{title}.md"
            filepath = os.path.join(self.vault_path, filename).replace('\\','/')

            # Formatting for labels and links
            tags = ' '.join([f"#{label} [[{label}]]" for label in labels])

            # Front matter (YAML headers)
            front_matter = '---\n'
            front_matter += f'title: {title}\n'
            front_matter += f'date: {datetime.now().strftime("%Y-%m-%d
%H:%M")}\n'
            front_matter += f'tags:\n'
            for label in labels:
                front_matter += f'  - {label}\n'
            front_matter += '---\n\n'

            # Content of the note
```

```
            note_content = front_matter + content + '\n\n' + tags

            # Saving note
            with open(filepath, 'w', encoding='utf-8') as f:
                f.write(note_content)

            self.logger.info(f"Note successfully saved: {filepath}")
        except Exception as e:
            self.logger.error(f"Error saving note: {e}")
            raise
```

A create_label_notes() metódus célja, hogy minden címkéhez különálló jegyzetet hozzon létre az Obsidian Vault-ban, amely tartalmazza a címke nevét és az összes kapcsolódó címkét. A folyamat először végigmegy az összes címkén, és minden címkéhez külön Markdown fájlt készít, amely tartalmazza a címke nevét. A fájlnevet a címke alapján hozza létre .md kiterjesztéssel, és meghatározza annak teljes elérési útvonalát, ami az Obsidian Vault és a Labels könyvtár kombinációja. Az előállított elérési útvonalat ezután ellenőrzi, hogy a Labels mappa létezik e. Amennyiben nem létezik, létrehozza azt. A jegyzet tartalmának összeállításakor a metódus először a címke nevét adja meg, mint a jegyzet címét, Markdown szintaxissal. Ha a címkének vannak más kapcsolódó címkéi, akkor egy ”Related Labels” szekciót is létrehoz a jegyzetben, ahol felsorolja az összes kapcsolódó címkét hivatkozásként. Miután összeállította a jegyzet teljes tartalmát, a metódus elmenti azt egy Markdown fájlba. Az elérési útvonalon létrehozza vagy frissíti a fájlt.

```
def create_label_notes(self, labels, connections):
        try:
            self.logger.info("Creating label notes in Obsidian...")
            for label in labels:
                filename = f"{label}.md"
                filepath = os.path.join(self.vault_path, 'Labels', filename)
                if not os.path.exists(os.path.dirname(filepath)):
                    os.makedirs(os.path.dirname(filepath))
                # Compiling the content of the note
                content = f'# {label}\n'
                related_labels = connections.get(label, [])
                if related_labels:
                    content += '\n## Related Labels\n'
                    for related_label in related_labels:
                        content += f'- [[{related_label}]]\n'
                # Save or update a note
                with open(filepath, 'w', encoding='utf-8') as f:
                    f.write(content)
                self.logger.debug(f"Label note created or updated: {filepath}")
            self.logger.info("Label notes successfully created or updated.")
        except Exception as e:
            self.logger.error(f"Error creating label notes: {e}")
            raise
```

4.2.5 label_generator_gpt.py

A GPTLabelGenerator osztály az OpenAI GPT modell felhasználásával címkék generálására szolgál. Az osztály célja, hogy a megadott dokumentum szövege alapján releváns címkéket hozzon létre, amelyeket később felhasználhatunk a dokumentumok

rendszerezésére, kereshetőségére és kapcsolataik kialakítására az Obsidian jegyzetalkalmazásban. Az osztály konstruktora beállítja a paraméterként kapott API kulcsot, mely biztosítja a hozzáférést az OpenAI szolgáltatásaihoz, lehetővé téve az API hívásokat, valamint a használandó GPT modell nevét. A modell nevének van egy alapértelemezett értéke, emiatt nem kötelező értékkel ellátni az osztály inicializálásakor, viszont későbbi fejlesztés során ezáltal módosítható.

```
1. class GPTLabelGenerator:
2.     def __init__(self, api_key=None, model_name='gpt-4o-mini'):
3.         self.logger = logging.getLogger(__name__)
4.         if api_key:
5.             self.client = OpenAI(api_key=api_key)
6.         else:
7.             self.client = OpenAI(os.getenv('OPENAI_API_KEY'))
8.
9.         if not self.client:
10.            self.logger.error("OpenAI API key is not set.")
11.            raise ValueError("OpenAI API key is not set.")
12.
13.        self.model_name = model_name
```

A GPTLabelGenerator osztályon belül található generate_labels() metódus célja, hogy címkéket generáljon a beolvasott szöveg alapján az OpenAI GPT modell használatával. A metódus egy előre definiált folyamatot követ annak érdekében, hogy a dokumentum tartalmát tekintve releváns címkék jöjjenek létre. Első lépésben a metódus létrehozza a promptot. A prompt egy szöveges utasítás, amely a GPT modell számára egyértelműen meghatározza, hogy mit várunk tőle. Ebben az esetben a prompt tartalmazza azt az instrukciót, hogy készítsen címkéket a megadott szöveg alapján, és vesszővel elválasztva jelenítse meg őket. Következő lépésként a metódus egy API hívást végez az OpenAI-hoz az openai.Completion.create() függvény segítségével. Ebben a lépésben a GPT modell elemzi a megadott szöveget, és címkéket generál hozzá. A generáláshoz a már korábban beállított GPT-4o-mini modellt használja. A paraméterek között szerepel a válaszban maximálisan felhasználható tokenek száma, amely 150-re van korlátozva, hogy egyszavas címkék kerüljenek létrehozásra, így lehetőséget adva a kicsit komplexebb szóösszetételű címkékre is. A temperature paraméter beállítása 0,3-ra biztosítja, hogy a válasz konzisztens és pontos legyen, minimális rugalmasság, kreativitás mellett. [25] Miután a GPT modell elkészítette a választ, a metódus kinyeri a generált címkéket a válaszból. Az OpenAI API válaszában található címkéket a choices[0].text segítségével érjük el, majd a strip() függvény segítségével eltávolítjuk az esetleges felesleges szóközöket. A címkék egy karakterláncként kerülnek visszaadásra, vesszővel elválasztva.

```
1. def generate_labels(self, text):
2.         try:
3.             self.logger.info("Generating labels using GPT model...")
4.             prompt = (
5.                 "Read the following text and generate relevant tags. "
6.                 "Provide the tags separated by commas.\n\nText:\n"
7.                 f"{text}\n\nTags:"
```

```
 8.             )
 9.             response = self.client.chat.completions.create(
10.                 model=self.model_name,
11.                 messages=[
12.                     {"role": "system", "content": "You are an intelligent assis-
tant that generates tags from text."},
13.                     {"role": "user", "content": prompt}
14.                 ],
15.                 max_tokens=150,
16.                 n=1,
17.                 temperature=0.3,
18.             )
19.             labels = response.choices[0].message.content.strip()
20.             self.logger.info("Labels successfully generated using GPT.")
21.             return labels
22.         except Exception as e:
23.             self.logger.error(f"Error during GPT label generation: {e}")
24.             return None
```

4.2.6 label_generator_llama.py

A LlamaLabelGenerator osztály a Llama nyelvi modellt használja címkék generálására. A Meta által fejlesztett Llama modell hasonló a GPT-hez, azonban specifikus esetekben más működési elveket és nyelvi struktúrákat ismer fel, valamint a GPT-vel ellentétben nem API hívásokkal működik, hanem lokálisan fut az alkalmazást futtató számítógépen. Ezzel lehetőséget adva annak az opciónak, hogy internet kapcsolat nélkül is használható legyen maga az alkalmazás. A LlamaLabelGenerator osztály inicializálásakor a konstruktor beállítja a modell nevét, melyet a .env fájlból nyer ki a program korábban mint környezeti változó. Ezen felül megvizsgálja, hogy a szükséges könyvtárak és illesztő programok telepítve vannak-e oly módon, hogy ellenőrzi van-e észlelhető GPU melyet a modell futtatására tud használni. Amennyiben nem talál erre alkalmas eszközt, úgy az alap eset lép érvénybe, ami azt jelenti, hogy a számítógép processzorát használja erre a célra. Ezt követően deklarálja a tokenizer és a model változókat, amiknek az értékeit az osztály saját load_model() metódusával állít be.

```
1. class LlamaLabelGenerator:
2.     def __init__(self, model_name):
3.         self.logger = logging.getLogger(__name__)
4.         self.model_name = model_name
5.         self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
6.         self.tokenizer = None
7.         self.model = None
8.         self.load_model()
```

A load_model() metódus a modell betöltéséért felel, hogy később a címkék generálásához fel lehessen használni. A metódus célja, hogy a Llama modell megfelelően be legyen töltve egy meghatározott eszközre, például CPU-ra vagy GPU-ra, attól függően, hogy milyen hardver áll rendelkezésre, valamint hogy a GPU használatához szükséges könyvtárak és illesztőprogramok telepítve vannak e az alkalmazást futtató számítógépen. Először a metódus az AutoTokenizer osztály segítségével betölti a tokenizálót, ami a nyelvi modell működésének alapját képezi, hiszen a tokenizáló feladata a bemeneti szöveg

feldarabolása olyan elemekre, amelyeket a modell értelmezni tud. A következő lépés a Llama nyelvi modell betöltése. Itt az AutoModelForCausalLM osztály segítségével a modell betöltéséhez megadjuk a modell nevét és hogy milyen eszközön futtassuk azt. A torch_dtype=torch.float16 paraméter azt biztosítja, hogy a modell betöltése kisebb precizitás mellett történjen, ami optimalizálja a memóriahasználatot. [26] Ennek eredményeképp a 3 milliárd paraméteres Llama modell képes futni kevesebb VRAM-on, például 8Gb-on is. A device_map paraméter pedig azt határozza meg, hogy a modell futtatása GPU-n vagy CPU-n történik, és ez automatikusan a rendelkezésre álló hardvertől függően kerül beállításra az osztály inicializálásakor.

```
def load_model(self):
        try:
            self.logger.info("Loading Llama tokenizer...")
            self.tokenizer = AutoTokenizer.from_pretrained(self.model_name)
            self.logger.info("Loading Llama model...")
            self.model = AutoModelForCausalLM.from_pretrained(
                self.model_name,
                torch_dtype=torch.float16,
                device_map=self.device,
            )
            self.logger.info("Llama model successfully loaded.")
        except Exception as e:
            self.logger.error(f"Error loading Llama model: {e}")
            raise
```

A generate_labels() metódus feladata, hogy egy adott szöveg alapján címkéket generáljon a LLaMA nyelvi modell segítségével. A metódus első lépésként létrehoz egy úgynevezett promptot, vagyis egy bemeneti utasítást, amelyet a LLaMA modellnek átad. Ez a prompt azért fontos, mert pontos iránymutatást ad a modell számára, és így biztosítja, hogy a generált válaszban címkéket kapjunk. Ezután a metódus tokenizálja a promptot. A tokenizálás során a szöveget számokká alakítja, amelyeket a modell értelmezni tud. Ezt a tokenizált adatot átmozgatja arra az eszközre, amelyet a device változóban meghatároztunk. Ezt követően a tokenizált bemenet alapján a metódus a modellt használja címkék generálására. A modell paramétereinek megadott értékek beállítják, hogy milyen hosszú legyen a válasz, hány választ generáljon, és hogy mekkora legyen a válaszok variabilitása. A do_sample paraméter igaz értéke miatt a modell valószínűségi mintavételezést alkalmaz a következő token kiválasztásakor. Ez azt jelenti, hogy a modell a következő szót valószínűségi eloszlás alapján választja ki. A hőmérséklet paraméter 0,3-ra van beállítva, mely biztosítja, hogy a válaszok konzisztensek maradnak kis kreativitással. [27] Amint a modell elkészítette a válaszát, a szövegen belül megkeresi a "Tags:" részt követő tartalmat, hogy ténylegesen csak a címkéket nyerje ki belőle. A metódus végül visszaadja a generált címkék listáját.

```
def generate_labels(self, text):
        try:
            self.logger.info("Generating labels using Llama model...")
            prompt = (
```

```
                "Read the following text and generate relevant tags. "
                "Provide the tags separated by commas.\n\nText:\n"
                f"{text}\n\nTags:"
            )
            inputs = self.tokenizer(prompt, return_tensors='pt').to(self.device)
            output = self.model.generate(
                **inputs,
                max_new_tokens=150,
                do_sample=True,
                temperature=0.3,
                eos_token_id=self.tokenizer.eos_token_id,
                pad_token_id=self.tokenizer.pad_token_id
            )
            generated_text = self.tokenizer.decode(output[0], skip_special_tokens=True)
            labels = generated_text.split("Tags:")[-1].strip()
            self.logger.info("Labels successfully generated using Llama.")
            return labels
        except Exception as e:
            self.logger.error(f"Error during Llama label generation: {e}")
            return None
```

4.2.7 preprocessing.py

A PreprocessingAgent osztály feladata, hogy előkészítse a szöveges adatokat a későbbi feldolgozás, elemzés vagy címkegenerálás számára. Az osztály inicializálásakor a konstruktor létrehoz egy logger objektumot, amelyet a későbbiekben használ a naplózásra.

```
class PreprocessingAgent:
    def __init__(self):
        self.logger = logging.getLogger(__name__)
```

A remove_punctuation() metódus célja, hogy a paraméterként kapott szövegből eltávolítsa az összes írásjelet, miközben megtartja a szöveg többi részét. Először is a metódus végig iterál a szöveg minden egyes karakterén, és az unicodedata.category() függvény segítségével az egyes karakterek Unicode kategóriáját azonosítja, amely alapján eldönthető, hogy az adott karakter írásjel-e vagy sem. Ha egy karakter írásjelként van azonosítva, azaz az Unicode kategóriája "P"-vel kezdődik, azt a metódus figyelmen kívül hagyja, és nem veszi be a végső karakterláncba. Végül visszaadja a tisztított szöveget.

```
def remove_punctuation(self, text):
        # Removes punctuation, but keeps accented letters
        text = ''.join(c for c in text if unicodedata.category(c)[0] != 'P')
        return text
```

A clean_text() metódus célja, hogy a szöveget teljes körűen előkészítse a címke generálása. A metódus több lépést is végrehajt, hogy a szöveg tiszta és jól formázott legyen, eltávolítva minden olyan elemet, amely zavarhatná a későbbi elemzést. Az feldolgozás első lépése a szöveg kisbetűssé alakítása. A metódus az összes karaktert kisbetűre alakítja, hogy a nyelvi modellek számára egységes formában kerüljön a bemeneti szöveg feldolgozásra. Ezt követően a metódus meghívja a korábban említett remove_punctuation()

metódust, amely eltávolítja az összes írásjelet a szövegből. Ezáltal a különféle írásjelek, mint például pontok, vesszők, felkiáltójelek nem zavarják a modell működését. A következő lépésben a metódus eltávolítja a szövegben található számokat. Ez azért lehet fontos, mert a számok gyakran nem relevánsak a címkék generálásnak szempontjából. Az utolsó feldolgozási lépés a felesleges szóközök eltávolítása és normalizálása. Ezt követően a tisztított szöveg visszaadásra kerül.

```
 1. def clean_text(self, text):
 2.         try:
 3.             self.logger.info("Preprocessing text...")
 4.             text = text.lower()
 5.             text = self.remove_punctuation(text)
 6.             text = re.sub(r'\d+', '', text)  # Remove numbers (if necessary)
 7.             text = re.sub(r'\s+', ' ', text)
 8.             self.logger.info("Text preprocessing completed.")
 9.             return text
10.         except Exception as e:
11.             self.logger.error(f"Error during text preprocessing: {e}")
12.             raise
```

4.2.8 document_loader_agent.py

A DocumentLoaderAgent osztálycélja, hogy különböző típusú dokumentumokból kivonja a szöveget. Az osztály felépítése moduláris, vagyis különböző fájltípusokhoz külön metódusokat tartalmaz, így egy adott dokumentumtípus esetén a legmegfelelőbb feldolgozási technikát alkalmazza. Az osztály inicializálásakor a konstruktor beállítja a feldolgozandó fájl elérési útvonalát, létrehoz egy üres String-et amelybe a beolvasott szöveg kerül majd, valamint beállítja a Tesseract OCR elérési útvonalát a pytesseract könyvtár számára, hogy tudja használni azt.

```
1. class DocumentLoaderAgent:
2.     def __init__(self, file_path):
3.         self.file_path = file_path
4.         self.text = ""
5.         self.logger = logging.getLogger(__name__)
6.         pytesseract.pytesseract.tesseract_cmd = r"C:\Program Files\Tesseract-
OCR\tesseract.exe"
```

A document_to_text() metódus célja, hogy különféle típusú dokumentumokat automatikusan felismerjen és a megfelelő metódus meghívásával kinyerje a szöveget. A metódus először ellenőrzi, hogy a fájl létezik-e, majd megállapítja a fájl kiterjesztését. Ehhez az os.path.splitext() függvényt használja, amely a fájl nevéből kiszűri a kiterjesztést. Ezt követően a fájl típusának megfelelő metódust hívja meg a beolvasáshoz.

```
 1. def document_to_text(self):
 2.         try:
 3.             if not os.path.exists(self.file_path):
 4.                 raise FileNotFoundError(f"The specified file does not exist:
{self.file_path}")
 5.
 6.             file_extension = os.path.splitext(self.file_path)[1].lower()
```

```
 7.
 8.             if file_extension == '.pdf':
 9.                 self.logger.info("Extracting text from PDF using PyPDF2...")
10.                 self.text = self.pdf_to_text()
11.             elif file_extension in ['.docx', '.doc']:
12.                 self.logger.info("Extracting text from Word document using
python-docx...")
13.                 self.text = self.word_to_text()
14.             elif file_extension == '.txt':
15.                 self.logger.info("Extracting text from txt document...")
16.                 self.text = self.text_to_text()
17.             elif file_extension in ['.png', '.jpg']:
18.                 self.logger.info("Extracting text from image document using
pytesseract...")
19.                 self.text = self.image_to_text()
20.             else:
21.                 raise ValueError(f"Unsupported file type: {file_extension}")
22.
23.             self.logger.info("Document successfully converted to text.")
24.             return self.text
25.         except Exception as e:
26.             self.logger.error(f"Error during document processing: {e}")
27.             raise
```

A pdf_to_text() metódus célja, hogy PDF dokumentumokból szöveget nyerjen ki. A folyamat első lépése, hogy megnyissa a PDF fájlt. Ehhez létrehoz egy PDF olvasót, amelynek segítségével hozzáfér az adott fájl oldalaihoz. A következő lépésben inicializál egy üres String változót, amelyet a későbbiekben feltölt az oldalakról kinyert szöveggel. Végig iterál a PDF összes oldalán, és minden egyes oldalt egyenként dolgoz fel, és a page.extract_text() függvénnyel próbál szöveget kinyerni. Ha egy oldalon található szöveg, azt hozzáfűzi az text változóhoz. Miután a metódus végigiterált az összes oldalon, és összegyűjtötte a szöveget, a feldolgozott dokumentum teljes tartalmát visszaadja.

```
1. def pdf_to_text(self):
2.         try:
3.             reader = PdfReader(self.file_path)
4.             text = ""
5.             for page_number, page in enumerate(reader.pages, start=1):
6.                 page_text = page.extract_text()
7.                 if page_text:
8.                     text += page_text
9.                 else:
10.                     self.logger.warning(f"No text found on page {page_number}.")
11.             return text
12.         except Exception as e:
13.             self.logger.error(f"Error during PDF processing: {e}")
14.             raise
```

A word_to_text() metódus célja, hogy egy .docx vagy .doc formátumú Word dokumentumból kivonja a szöveget. Először megnyitja a Word dokumentumot a megadott fájl elérési útvonala alapján. Ehhez a python-docx könyvtárat használja, amely lehetővé teszi a .docx formátumú Word fájlok feldolgozását. A következő lépésben a metódus létrehoz egy üres szövegváltozót, amelybe a dokumentumból kinyert szöveget fogja tárolni. Ezután a metódus végigmegy a Word dokumentum összes bekezdésén, és hozzáadja a text

változóhoz oly módon, hogy az egyes bekezdések közötti elválasztás megmaradjon. Miután a metódus végigment a dokumentum összes bekezdésén, visszaadja a beolvasott szöveget.

```
 1. def word_to_text(self):
 2.         try:
 3.             doc = docx.Document(self.file_path)
 4.             text = ""
 5.             for para in doc.paragraphs:
 6.                 text += para.text + '\n'
 7.             return text
 8.         except Exception as e:
 9.             self.logger.error(f"Error during Word document processing: {e}")
10.             raise
```

A text_to_text() metódus feladata, hogy egy egyszerű .txt formátumú szövegfájlból kiolvassa a szöveget és visszaadja azt. Amikor a text_to_text() metódus elindul, először megpróbálja megnyitni a megadott fájlt. Ehhez az open() függvényt használja. A fájl megnyitásakor a metódus az UTF-8 kódolást használja, mely az egyik legelterjedtebb kódolás. Miután sikeresen megnyitotta a fájlt, a metódus a read() hívással kiolvassa a fájl teljes tartalmát, melyet egyetlen szöveglánckén ad vissza. Amikor a szöveg beolvasása befejeződött, a metódus egyszerűen visszaadja a kinyert szöveget.

```
1. def text_to_text(self):
2.         try:
3.             with open(self.file_path, 'r', encoding='utf-8') as f:
4.                 text = f.read()
5.             return text
6.         except Exception as e:
7.             self.logger.error(f"Error during text file processing: {e}")
8.             raise
```

Az image_to_text() metódus feladata, hogy képfájlokból kinyerje a szöveget OCR technológia segítségével. Ezáltal lehetővé teszi, hogy a .png vagy .jpg formátumú képfájlokban található szöveges információ szövegként legyen elérhető. A metódus első feladata a kép megnyitása. Ehhez a PIL könyvtár Image.open() függvényét használja, amely lehetővé teszi a megadott elérési útvonalon található képfájl megnyitását. Miután a képet sikeresen megnyitotta, a metódus a pytesseract.image_to_string() függvényt hívja meg, amely a megadott képből próbálja meg kinyerni a szöveget. Amikor sikeresen kinyeri a szöveget a képből, a metódus ezt a szöveget elmenti, majd visszaadja.

```
1. def image_to_text(self):
2.         try:
3.             text = pytesseract.image_to_string(Image.open(self.file_path),
lang='eng')
4.             return text
5.         except Exception as e:
6.             self.logger.error(f"Error during image file processing: {e}")
7.             raise
```

5. Tesztelés

A program tesztelése nehéz feladat. Az LLM-ek által generált címkék eredményeinek mérésére hivatalos tesztelést, minősítést nem találtam, így a felhasználói hibák tesztelésére került sor. A tesztelés során először a betölteni kívánt dokumentumok fájlformátumainak feldolgozhatósága került vizsgálatra. A PDF, Word, txt és kép formátumok mellett egyéb szöveg tárolására alkalmas fájlformátum is a kijelölt mappába tartalmát alkotta.

Dokumentum formátuma	**Várt eredmény**	**Eredmény**
PDF	sikeres	sikeres
TXT	sikeres	sikeres
DOCX	sikeres	sikeres
PNG	sikeres	sikeres
JPG	sikeres	sikeres
JSON	sikertelen	sikertelen
XML	sikertelen	sikertelen
HTML	sikertelen	sikertelen

A különböző fájlformátumok tesztelése mellett a felhasználótól elvárt bemenet tesztelése tűnt szükségesnek. Ebben a bemenetben a felhasználó beírja, hogy milyen modellt kíván használni a kijelölt mappában lévé dokumentumok feldolgozására.

Felhasználói bemenet	**Várt eredmény**	**Eredmény**
gpt	gpt	gpt
GPT	gpt	gpt
gPt	gpt	gpt
.GPT	sikertelen	sikertelen
gépété	sikertelen	sikertelen
gpt (szóközök előtte és utána)	gpt	gpt
llama	llama	llama
Llama	llama	llama
LLAMA	llama	llama
lama	sikertelen	sikeretlen
.llama.	sikertelen	sikertelen
llama (szóközök előtte és utána)	llama	llama
xyz	sikertelen	sikertelen

A tesztelés során az alkalmazás az elvárt működést produkálta. Sikeresen beolvasta a PDF, Word, txt és kép formátumú fájlokat, és nyerte ki a szöveges tartalmat belőlük. A nem támogatott formátumú fájlokat nem tudta beolvasni, mivel azok implementálására nem került sor a fejlesztés során. A felhasználótól várt bemenet tesztelése során az elvárt karakterlánc kis és nagybetűs alakja nem okozott hibát mivel a kódban egyből normalizálást hajtunk végre rajta, vagyis kibetűs formára alakítunk minden karaktert. Emellett a karakterlánc elején és végén található esetleges szóközök is eltávolításra kerülnek.

6. Továbbfejlesztési lehetőségek

Az alkalmazás fejlesztése és tervezése során több olyan lehetőség, ötlet is felmerült, melyek növelhetik a rendszer funkcionalitását, hatékonyságát:

- Több nyelvű feldolgozás bevezetése: Ugyan a GPT 4o-mini alkalmas több nyelven való feldolgozásra, a Llama és a Spacy modell esetében más vagy nagyobb modelleket igényelnek. Ez az opció kivitelezhető lehet akár egy BERT modell alkalmazásával is.
- Frontend felület: Jelenleg az alkalmazás főként backend fókuszú, és az Obsidian integrációra épít. A felhasználói élmény javítása érdekében lehetőség lenne egy saját frontend fejlesztésére, ahol a felhasználó nem parancssorral kommunikál az alkalmazással.
- Generált címkék tartalmának hozzáfűzése: A címkék generálása során az alkalmazás jelenleg csak kapcsolatokat határozza meg az egyes címkék között. A jobb tudásháló kialakítása érdekében továbbfejleszthető a címke generálás oly módon, hogy a tartalmat, mely alapján generálta az adott címkét, a kapcsolatok felsorolása után a címke tartalmaként belemásolja, bele illessze.
- Automatikus modellválasztás: Az alkalmazás mostani állapotában a felhasználónak manuálisan kell kiválasztania, hogy a GPT vagy a Llama modell segítségével történjen-e a címkézés. Fejlesztési lehetőségként érdemes lehet megvalósítani egy automatikus modellválasztási algoritmust, mely előre megadott kritériumok alapján dönti el, hogy mely modell alkalmazása lenne a legalkalmasabb az adott dokumentumhoz.
- Docker konténerizáció: A hordozhatóság és a használhatóság egyszerűségének fejlesztése érdekében célszerű konténerizálni a programot. Jelen állapotában a program felállítása és beüzemelése perceket is igénybe vehet, valamint relatíve komplexnek tekinthető, így a docker konténerizálás minőségi fejlesztést jelent. Docker konténerizálás fejlesztése megkezdődött, viszont a határidő közeledtével nem sikerült befejezni.
- Több fájlformátum támogatása: Az alkalmazás fejlesztése során csak a hétköznapi életben leggyakrabban előforduló formátumok feldolgozása került implementálásra. Ezen támogatott formátumok listájának kibővítése hasznos fejlesztés olyan esetekben, ahol az információk más struktúrában vannak tárolva.

7. Összefoglaló magyar nyelven

A szakdolgozat keretében egy teljes backend alkalmazást készítettem, amely az OpenAI GPT, Llama modell és Obsidian alkalmazás segítségével valósul meg. A rendszer célja, hogy PDF, Word, txt- és képformátumú dokumentumokat dolgozzon fel, majd azokból releváns címkéket generáljon, amelyek segítik a jegyzetelést és a kapcsolatok felépítését az Obsidian tudásbázis alkalmazásban.

Az alkalmazás fő komponense egy Python alapú backend, amely a MainAgent segítségével hajtja végre a dokumentumfeldolgozást és címkézést. A rendszer a megadott fájlokat különféle formátumban dolgozza fel a DocumentLoaderAgent révén, amely OCR technológiát is alkalmaz a szöveg kinyerésére. Az így nyert szöveget a PreprocessingAgent tisztítja meg, például eltávolítja a felesleges írásjeleket és normalizálja a szöveget.

A címkék generálása az OpenAI GPT modellje vagy a Llama modell segítségével történik, amelyek közül a felhasználó választhatja ki a kívánt eszközt. A GPTLabelGenerator és a LlamaLabelGenerator a feldolgozott szöveg alapján címkéket hoz létre, amelyek segítenek a dokumentumok könnyebb rendszerezésében. A címkék további normalizálása és összekapcsolása a Spacy segítségével megy végbe, lehetővé téve a címkék közötti kapcsolatok felismerését és a tudásháló felépítését.

A generált címkék és a szövegből kinyert információk az Obsidian vaultba kerülnek mentésre. Az Obsidian segítségével a ObsidianAgent biztosítja, hogy a dokumentumok tartalmához kapcsolódó címkék és azok kapcsolatai megfelelően kerüljenek tárolásra, így a felhasználó könnyedén navigálhat a különböző dokumentumok között és kapcsolódó információk alapján építhet tudáshálózatot.

A rendszer beállítása és futtatása során a felhasználónak létre kell hoznia egy virtuális környezetet, amely tartalmazza az összes szükséges függőséget, továbbá szükség van egy .env fájlra, amely tartalmazza a szükséges API kulcsokat és modellek neveit. A program használatával a dokumentumok hatékony feldolgozására és rendszerezésére van lehetőség, elősegítve az információszervezés és jegyzetelés egyszerűsítését és hatékonyságát.

8. Sumarry in english

In this thesis I created a complete backend application using the OpenAI GPT, Llama model and Obsidian application. The system is designed to process documents in PDF, Word, txt and image formats and generate relevant tags from them to assist annotation and link building in the Obsidian knowledge base application.

The main component of the application is a Python-based backend that performs document processing and tagging with the help of MainAgent. The system processes the given files in different formats through DocumentLoaderAgent, which also uses OCR technologic to extract the text. The resulting text is then processed by the PreprocessingAgent, for example by removing unnecessary punctuation and normalising the text.

The tags are generated using the OpenAI GPT model or the Llama model, from which the user can choose the desired device. The GPTLabelGenerator and LlamaLabelGenerator generate labels based on the processed text, which help to organise documents more easily. Further normalisation and linking of labels is performed by Spacy, enabling the recognition of relationships between labels and the building of a knowledge network.

The tags generated and the information extracted from the text are sent to the Obsidian Vault. With Obsidian, ObsidianAgent ensures that the tags and their relationships to the content of documents are properly stored, allowing the user to easily navigate between different documents and build a knowledge network based on related information.

To set up and run the system, the user needs to create a virtual environment containing all the necessary dependencies and an .env file containing the necessary API keys and model names. Using the program, it is possible to process and organise documents efficiently, helping to simplify and increase the efficiency of information organisation and annotation.

9. Köszönetnyilvánítás

Szeretnék köszönetet mondani Szigeti Mark Bencének, aki vállalta a konzulensi szerepet a szakdolgozatom megírásához, és azonnal segítséget nyújtott minden felmerülő kérdésemmel kapcsolatban.

Szeretném megköszönni a családtagjaimnak és a barátaimnak a belém fektetett hitüket, bizalmukat, és hogy végig támogattak a tanulmányaim során.

10. Irodalomjegyzék

[1] IBM Blog, „What Is Optical Character Recognition (OCR)?," IBM, 05 01 2022. [Online]. Available: https://www.ibm.com/blog/optical-character-recognition/. [Hozzáférés dátuma: 26 02 2024].

[2] TechTarget, „OCR (optical character recognition)," TechTarget, 11 2022. [Online]. Available: https://www.techtarget.com/searchcontentmanagement/definition/OCR-optical-character-recognition. [Hozzáférés dátuma: 26 02 2024].

[3] AWS, „What is OCR (Optical Character Recognition)?," Amazon, [Online]. Available: https://aws.amazon.com/what-is/ocr/. [Hozzáférés dátuma: 27 02 2024].

[4] S. Reddy, „What is an OCR ??," Towards Data Science, 25 03 2019. [Online]. Available: https://towardsdatascience.com/what-is-ocr-7d46dc419eb9. [Hozzáférés dátuma: 27 02 2024].

[5] Coursera Staff, „What Is Metadata?," Coursera, 29 11 2023. [Online]. Available: https://www.coursera.org/articles/what-is-metadata?. [Hozzáférés dátuma: 01 03 2024].

[6] Imperva, „Metadata," Imperva, [Online]. Available: https://www.imperva.com/learn/data-security/metadata/. [Hozzáférés dátuma: 02 03 2024].

[7] G. Kranz, „metadata," TechTarget, 07 2021. [Online]. Available: https://www.techtarget.com/whatis/definition/metadata. [Hozzáférés dátuma: 02 03 2024].

[8] L. d. Leyritz, „What is metadata? - Benefits and Examples," CastorDoc, 09 05 2023. [Online]. Available: https://www.castordoc.com/blog/what-is-metadata. [Hozzáférés dátuma: 04 03 2024].

[9] Coursera Staff, „What Is Metadata Management?," Coursera, 09 04 2024. [Online]. Available: https://www.coursera.org/articles/metadata-management?. [Hozzáférés dátuma: 02 05 2024].

[10] Coursera Staff, „What is Natural Language Processing? Definition and Examples," Coursera, 19 03 2024. [Online]. Available: https://www.coursera.org/articles/natural-language-processing. [Hozzáférés dátuma: 11 04 2024].

[11] IBM, „What is natural language processing (NLP)?,” IBM, [Online]. Available: https://www.ibm.com/topics/natural-language-processing. [Hozzáférés dátuma: 11 04 2024].

[12] BME, „Neurális hálók,” BME, [Online]. Available: http://project.mit.bme.hu/mi_almanach/books/aima/ch20s05. [Hozzáférés dátuma: 16 04 2024].

[13] R. Farkas, „MÉLY GÉPI TANULÁS (DEEP LEARNING),” Szegedi Tudományegyetem, [Online]. Available: https://www.inf.u-szeged.hu/~rfarkas/ML20/deep_learning. [Hozzáférés dátuma: 16 04 2024].

[14] Spotfire, „What is a neural network?,” Cloud Software Group, Inc., [Online]. Available: https://www.spotfire.com/glossary/what-is-a-neural-network. [Hozzáférés dátuma: 16 04 2024].

[15] NVIDIA, „Generative AI,” NVIDIA Corporation, [Online]. Available: https://www.nvidia.com/en-us/glossary/generative-ai/. [Hozzáférés dátuma: 02 05 2024].

[16] M. S. Cole Stryker, „What is generative AI?,” IBM, 22 03 2024. [Online]. Available: https://www.ibm.com/topics/generative-ai. [Hozzáférés dátuma: 02 05 2024].

[17] A. Porter, „Unveiling 6 Types of Generative AI,” BigID, 15 08 2023. [Online]. Available: https://bigid.com/blog/unveiling-6-types-of-generative-ai/. [Hozzáférés dátuma: 2 05 2024].

[18] S. Singh, „Top methods to evaluate the performance of deep learning models,” Labellerr, 23 11 2022. [Online]. Available: https://www.labellerr.com/blog/evaluate-the-performance-of-deep-learning-models/. [Hozzáférés dátuma: 20 04 2024].

[19] S. K. Agrawal, „Metrics to Evaluate your Classification Model to take the right decisions,” Analytics Vidhya, 15 02 2024. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/07/metrics-to-evaluate-your-classification-model-to-take-the-right-decisions/. [Hozzáférés dátuma: 20 04 2024].

[20] Docker, „Docker Overview,” Docker Inc., [Online]. Available: https://docs.docker.com/get-started/overview/. [Hozzáférés dátuma: 30 03 2024].

[21] Docker, „Use containers to Build, Share and Run your applications,” Docker Inc., [Online]. Available: https://www.docker.com/resources/what-container/. [Hozzáférés dátuma: 30 03 2024].

[22] ITHub, „Docker: bevezetés a containerek világába,” ITHub, 30 10 2017. [Online]. Available: https://ithub.hu/blog/post/Docker_bevezetes_a_containerek_vilagaba. [Hozzáférés dátuma: 30 03 2024].

[23] A. Das, „Containerization: A Beginner's Guide to its Impact on Software Development,” DEV Community, 07 02 2023. [Online]. Available: https://dev.to/bitohq/containerization-a-beginners-guide-to-its-impact-on-software-development-280g. [Hozzáférés dátuma: 31 03 2024].

[24] GitHub, „What is containerization?,” GitHub Inc., 23 05 2022. [Online]. Available: https://resources.github.com/devops/containerization/. [Hozzáférés dátuma: 31 03 2024].

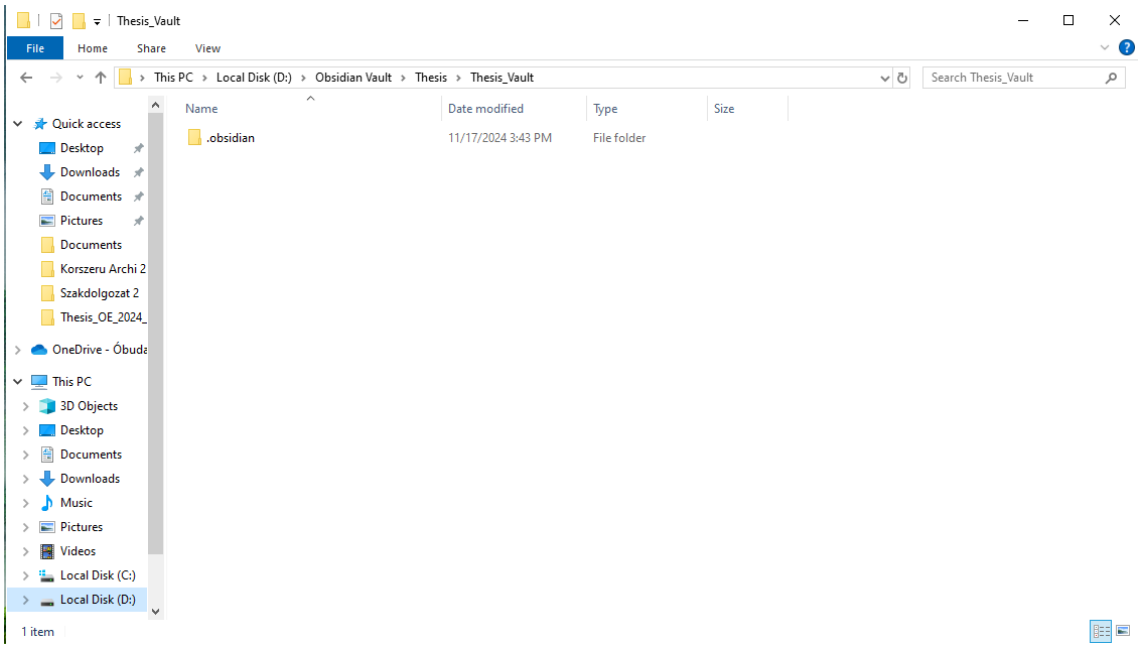
[25] OpenAI, „OpenAI Platform,” OpenAI, [Online]. Available: https://platform.openai.com/docs/api-reference/audio/createTranslation. [Hozzáférés dátuma: 24 11 2024].

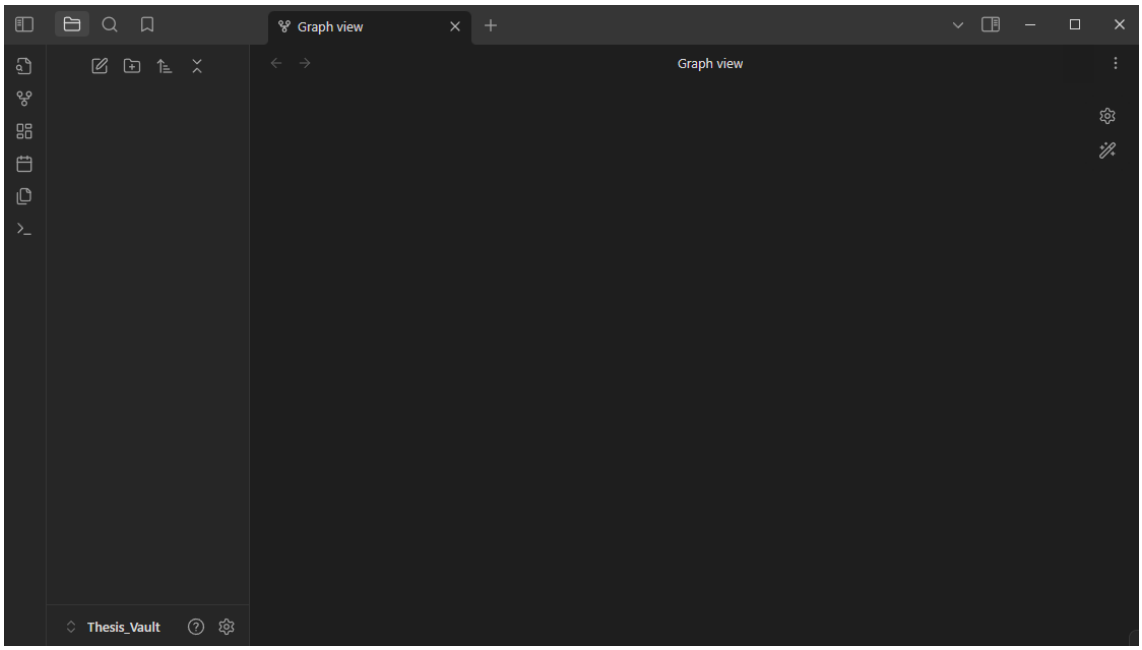
[26] H. face, „Hugging face Auto Classes,” Hugging face, [Online]. Available: https://huggingface.co/docs/transformers/model_doc/auto. [Hozzáférés dátuma: 24 11 2024].

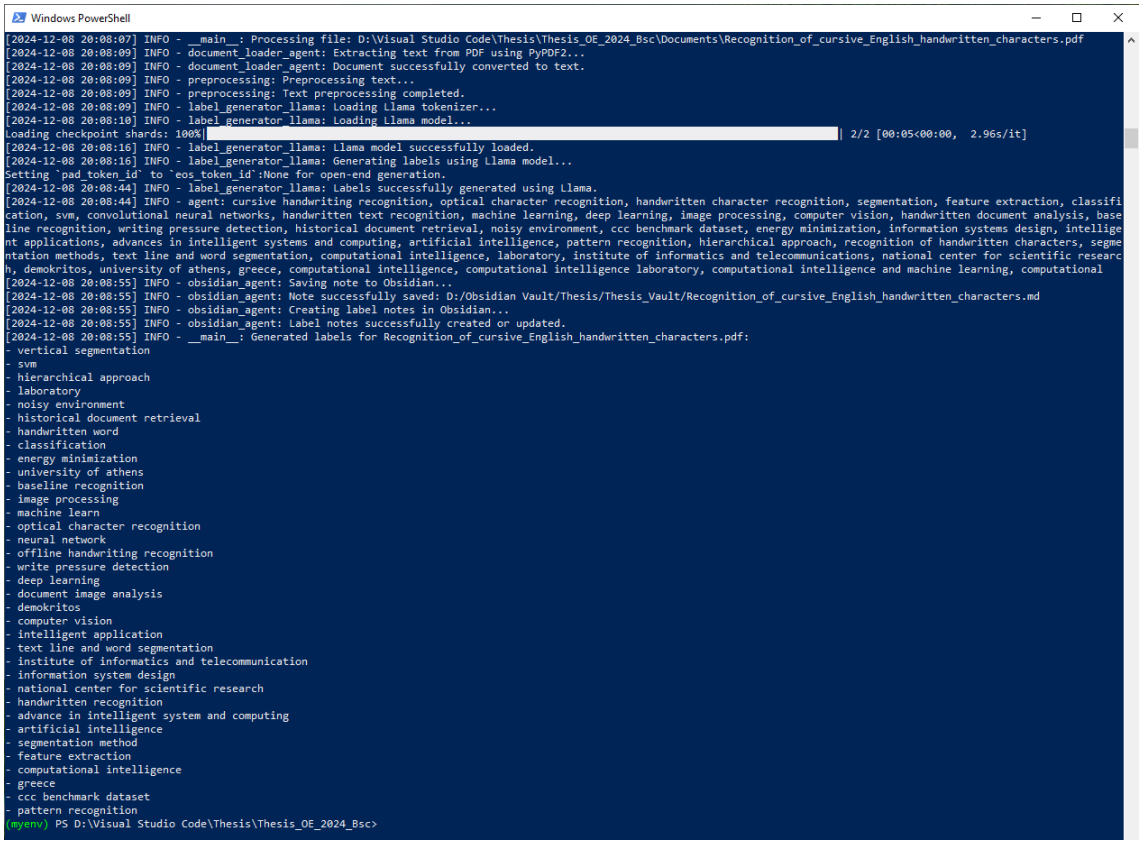
[27] H. face, „Hugging face Transformers,” Hugging face, [Online]. Available: https://huggingface.co/docs/transformers/main_classes/text_generation. [Hozzáférés dátuma: 20 11 2024].

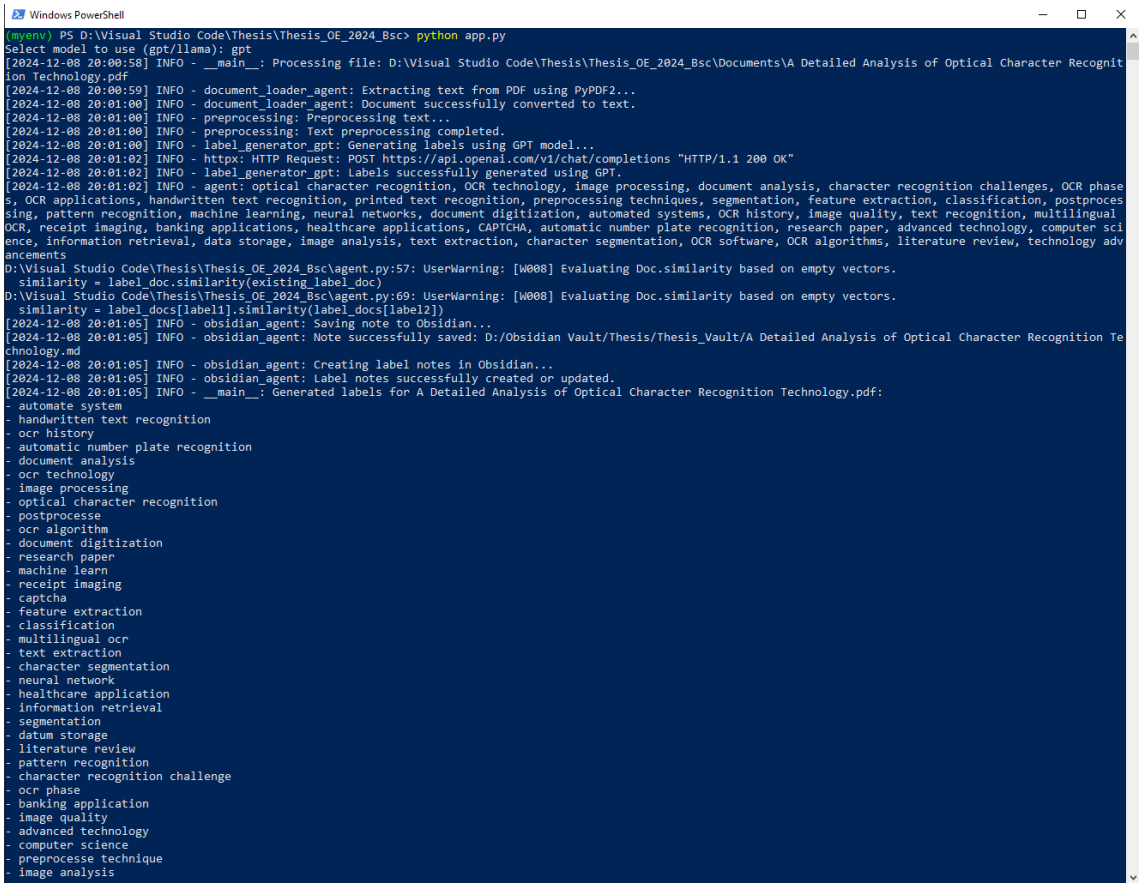
11. Ábrajegyzék

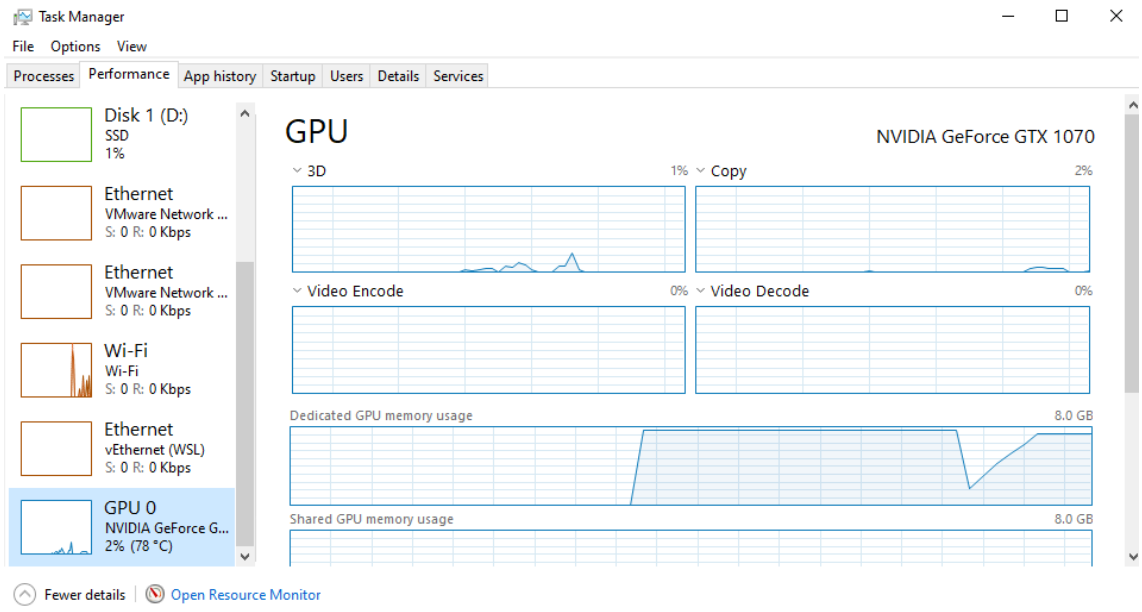
12. Mellékletek

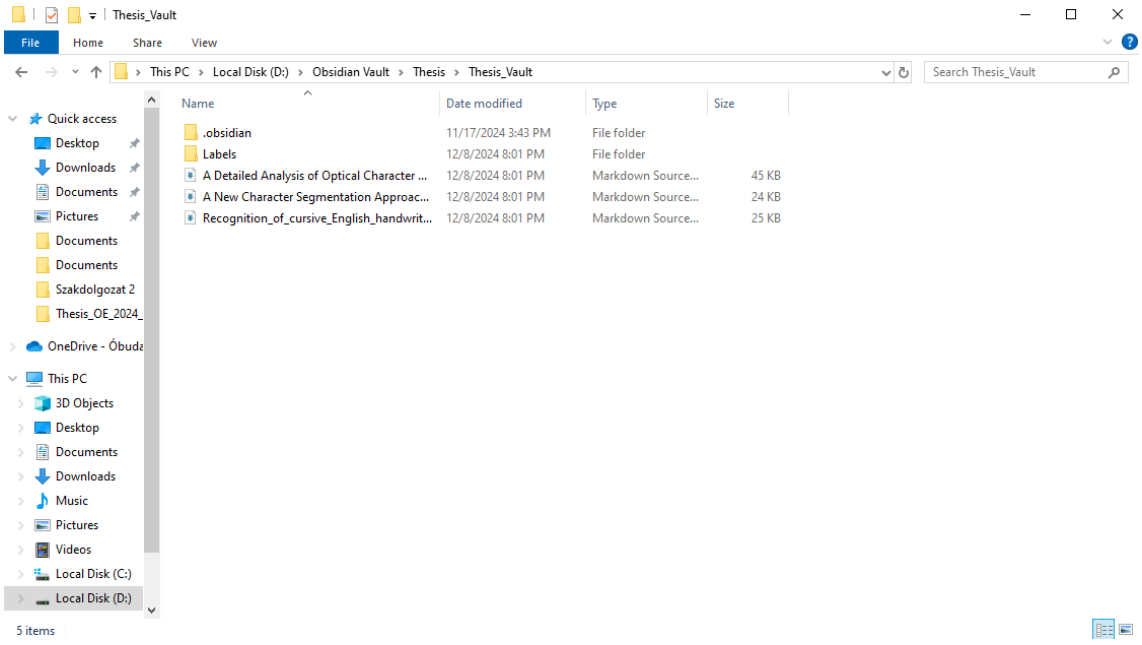


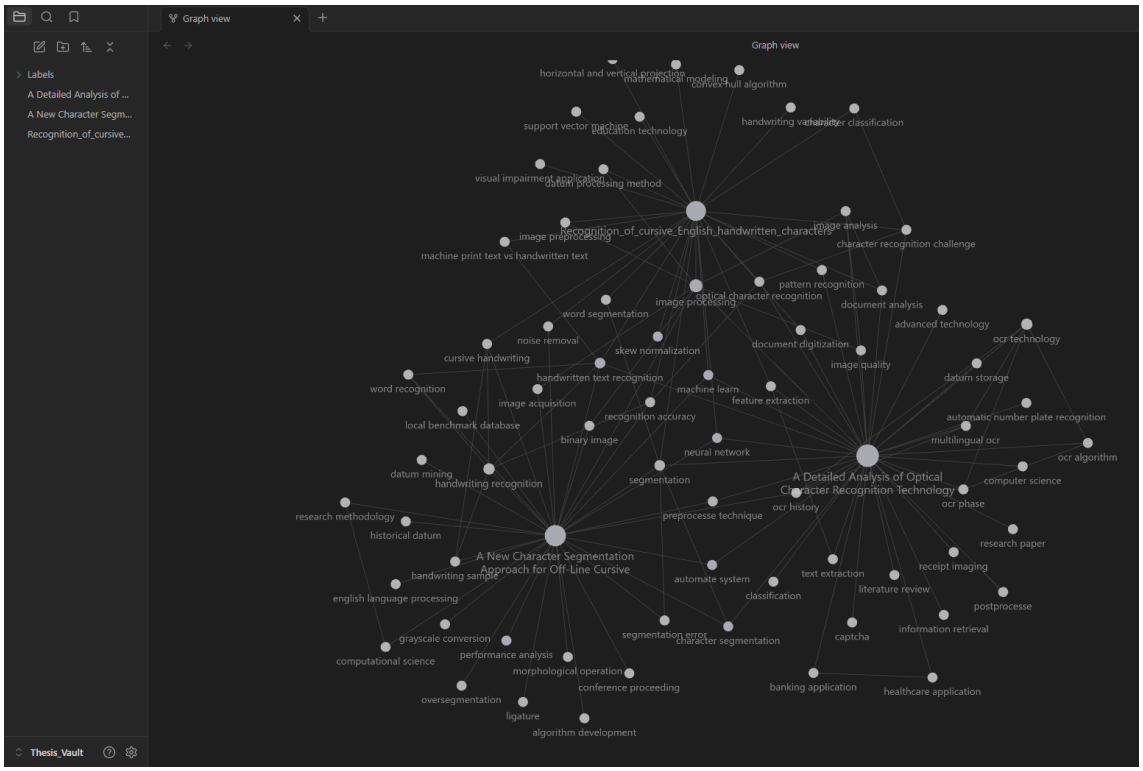


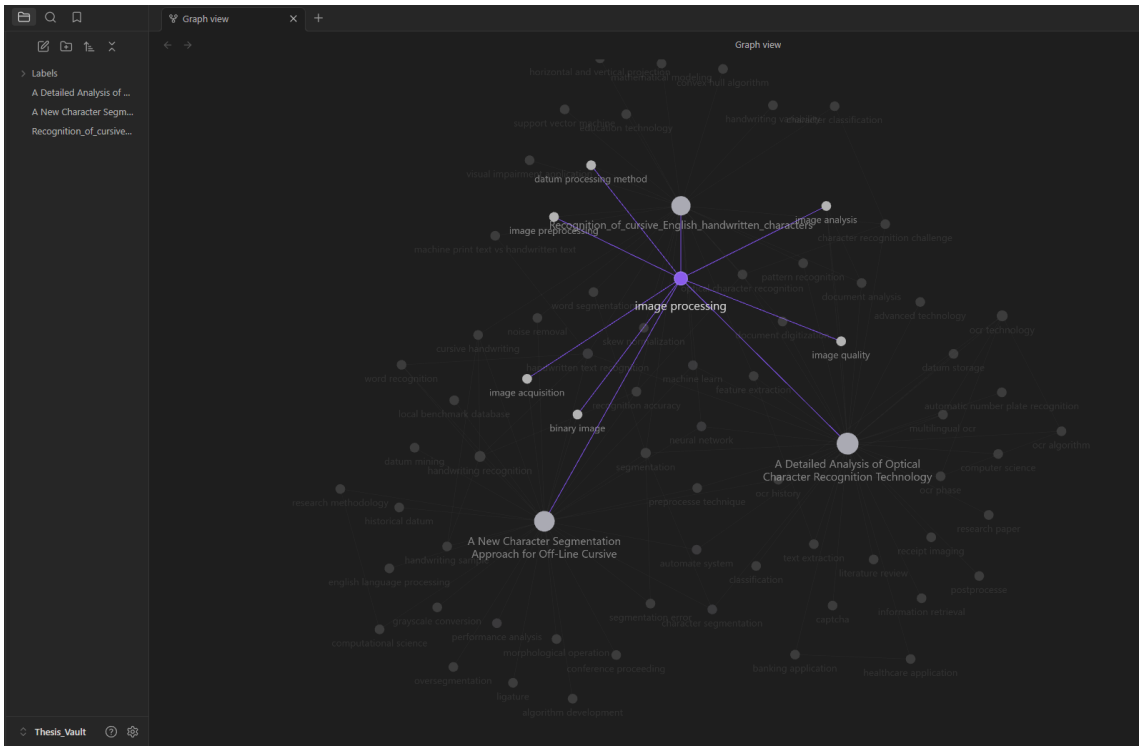

Windows PowerShell
[2024-12-08 20:08:07] INFO - __main__: Processing file: D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc\Documents\Recognition_of_cursive_English_handwritten_characters.pdf
[2024-12-08 20:08:09] INFO - document_loader_agent: Extracting text from PDF using PyPDF2...
[2024-12-08 20:08:09] INFO - document_loader_agent: Document successfully converted to text.
[2024-12-08 20:08:09] INFO - preprocessing: Preprocessing text...
[2024-12-08 20:08:09] INFO - preprocessing: Text preprocessing completed.
[2024-12-08 20:08:09] INFO - label_generator_llama: Loading Llama tokenizer...
[2024-12-08 20:08:10] INFO - label_generator_llama: Loading Llama model...
Loading checkpoint shards: 100% 2/2 [00:05<00:00, 2.96s/it]
[2024-12-08 20:08:16] INFO - label_generator_llama: Llama model successfully loaded.
[2024-12-08 20:08:16] INFO - label_generator_llama: Generating labels using Llama model...
Setting `pad_token_id` to `eos_token_id`:None for open-end generation.
[2024-12-08 20:08:44] INFO - label_generator_llama: Labels successfully generated using Llama.
[2024-12-08 20:08:44] INFO - agent: cursive handwriting recognition, optical character recognition, handwritten character recognition, segmentation, feature extraction, classification, svm, convolutional neural networks, handwritten text recognition, machine learning, deep learning, image processing, computer vision, handwritten document analysis, baseline recognition, writing pressure detection, historical document retrieval, noisy environment, ccc benchmark dataset, energy minimization, information systems design, intelligent applications, advances in intelligent systems and computing, artificial intelligence, pattern recognition, hierarchical approach, recognition of handwritten characters, segmentation methods, text line and word segmentation, computational intelligence, laboratory, institute of informatics and telecommunications, national center for scientific research, demokritos, university of athens, greece, computational intelligence, computational intelligence laboratory, computational intelligence and machine learning, computational
[2024-12-08 20:08:55] INFO - obsidian_agent: Saving note to Obsidian...
[2024-12-08 20:08:55] INFO - obsidian_agent: Note successfully saved: D:/Obsidian Vault/Thesis/Thesis_Vault/Recognition_of_cursive_English_handwritten_characters.md
[2024-12-08 20:08:55] INFO - obsidian_agent: Creating label notes in Obsidian...
[2024-12-08 20:08:55] INFO - obsidian_agent: Label notes successfully created or updated.
[2024-12-08 20:08:55] INFO - __main__: Generated labels for Recognition_of_cursive_English_handwritten_characters.pdf:
- vertical segmentation
- svm
- hierarchical approach
- laboratory
- noisy environment
- historical document retrieval
- handwritten word
- classification
- energy minimization
- university of athens
- baseline recognition
- image processing
- machine learn
- optical character recognition
- neural network
- offline handwriting recognition
- write pressure detection
- deep learning
- document image analysis
- demokritos
- computer vision
- intelligent application
- text line and word segmentation
- institute of informatics and telecommunication
- information system design
- national center for scientific research
- handwritten recognition
- advance in intelligent system and computing
- artificial intelligence
- segmentation method
- feature extraction
- computational intelligence
- greece
- ccc benchmark dataset
- pattern recognition
(myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc>

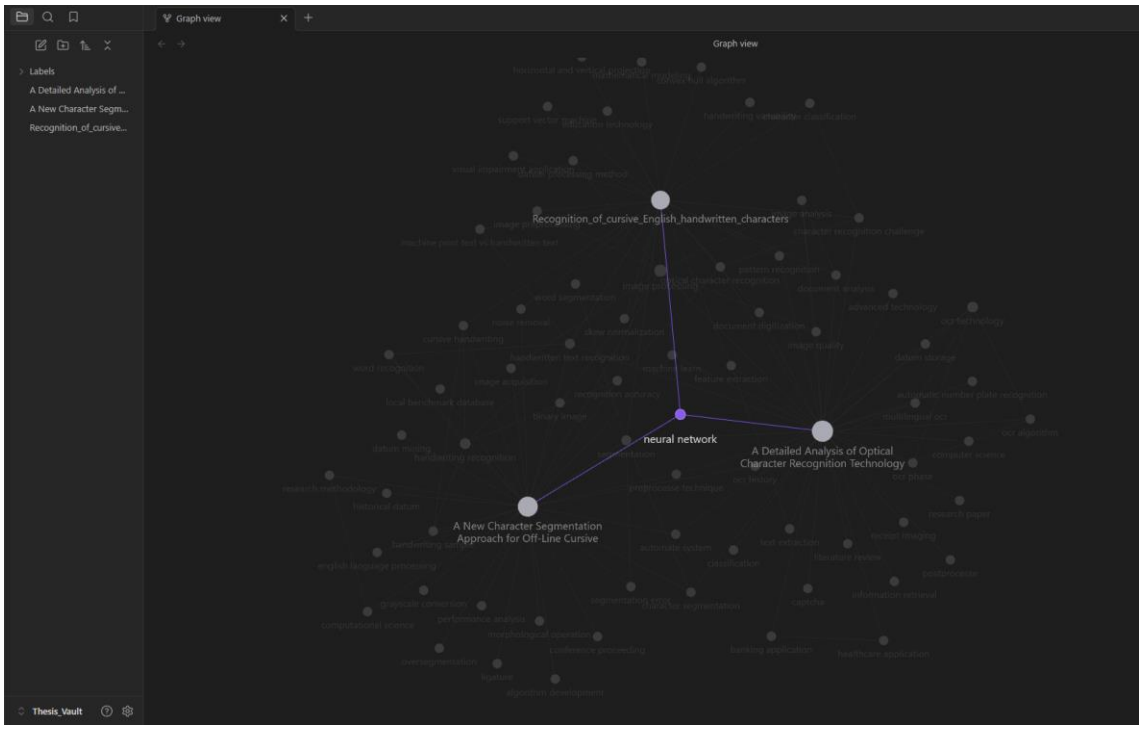

Windows PowerShell
(myenv) PS D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc> python app.py
Select model to use (gpt/llama): gpt
[2024-12-08 20:00:58] INFO - __main__: Processing file: D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc\Documents\A Detailed Analysis of Optical Character Recognition Technology.pdf
[2024-12-08 20:00:59] INFO - document_loader_agent: Extracting text from PDF using PyPDF2...
[2024-12-08 20:01:00] INFO - document_loader_agent: Document successfully converted to text.
[2024-12-08 20:01:00] INFO - preprocessing: Preprocessing text...
[2024-12-08 20:01:00] INFO - preprocessing: Text preprocessing completed.
[2024-12-08 20:01:00] INFO - label_generator_gpt: Generating labels using GPT model...
[2024-12-08 20:01:02] INFO - httpx: HTTP Request: POST https://api.openai.com/v1/chat/completions "HTTP/1.1 200 OK"
[2024-12-08 20:01:02] INFO - label_generator_gpt: Labels successfully generated using GPT.
[2024-12-08 20:01:02] INFO - agent: optical character recognition, OCR technology, image processing, document analysis, character recognition challenges, OCR phases, OCR applications, handwritten text recognition, printed text recognition, preprocessing techniques, segmentation, feature extraction, classification, postprocessing, pattern recognition, machine learning, neural networks, document digitization, automated systems, OCR history, image quality, text recognition, multilingual OCR, receipt imaging, banking applications, healthcare applications, CAPTCHA, automatic number plate recognition, research paper, advanced technology, computer science, information retrieval, data storage, image analysis, text extraction, character segmentation, OCR software, OCR algorithms, literature review, technology advancements
D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc\agent.py:57: UserWarning: [W008] Evaluating Doc.similarity based on empty vectors.
similarity = label_doc.similarity(existing_label_doc)
D:\Visual Studio Code\Thesis\Thesis_OE_2024_Bsc\agent.py:69: UserWarning: [W008] Evaluating Doc.similarity based on empty vectors.
similarity = label_docs[label1].similarity(label_docs[label2])
[2024-12-08 20:01:05] INFO - obsidian_agent: Saving note to Obsidian...
[2024-12-08 20:01:05] INFO - obsidian_agent: Note successfully saved: D:/Obsidian Vault/Thesis/Thesis_Vault/A Detailed Analysis of Optical Character Recognition Technology.md
[2024-12-08 20:01:05] INFO - obsidian_agent: Creating label notes in Obsidian...
[2024-12-08 20:01:05] INFO - obsidian_agent: Label notes successfully created or updated.
[2024-12-08 20:01:05] INFO - __main__: Generated labels for A Detailed Analysis of Optical Character Recognition Technology.pdf:
- automate system
- handwritten text recognition
- ocr history
- automatic number plate recognition
- document analysis
- ocr technology
- image processing
- optical character recognition
- postprocesse
- ocr algorithm
- document digitization
- research paper
- machine learn
- receipt imaging
- captcha
- feature extraction
- classification
- multilingual ocr
- text extraction
- character segmentation
- neural network
- healthcare application
- information retrieval
- segmentation
- datum storage
- literature review
- pattern recognition
- character recognition challenge
- ocr phase
- banking application
- image quality
- advanced technology
- computer science
- preprocesse technique
- image analysis


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Disk 1 (D:)
SSD
1%
Ethernet
VMware Network ...
S: 0 R: 0 Kbps
Ethernet
VMware Network ...
S: 0 R: 0 Kbps
Wi-Fi
Wi-Fi
S: 0 R: 0 Kbps
Ethernet
vEthernet (WSL)
S: 0 R: 0 Kbps
GPU 0
NVIDIA GeForce G...
2% (78 °C)
GPU
NVIDIA GeForce GTX 1070
3D 1%
Copy 2%
Video Encode 0%
Video Decode 0%
Dedicated GPU memory usage 8.0 GB
Shared GPU memory usage 8.0 GB
Fewer details
Open Resource Monitor


Thesis_Vault
File Home Share View
This PC › Local Disk (D:) › Obsidian Vault › Thesis › Thesis_Vault
Search Thesis_Vault
Quick access
Desktop
Downloads
Documents
Pictures
Documents
Documents
Szakdolgozat 2
Thesis_OE_2024_
OneDrive - Óbuda
This PC
3D Objects
Desktop
Documents
Downloads
Music
Pictures
Videos
Local Disk (C:)
Local Disk (D:)
Name
Date modified
Type
Size
.obsidian 11/17/2024 3:43 PM File folder
Labels 12/8/2024 8:01 PM File folder
A Detailed Analysis of Optical Character ... 12/8/2024 8:01 PM Markdown Source... 45 KB
A New Character Segmentation Approac... 12/8/2024 8:01 PM Markdown Source... 24 KB
Recognition_of_cursive_English_handwrit... 12/8/2024 8:01 PM Markdown Source... 25 KB
5 items


Graph view
Labels
A Detailed Analysis of ...
A New Character Segm...
Recognition_of_cursive...
Thesis_Vault
horizontal and vertical projection
mathematical modeling
convex hull algorithm
handwriting variability
character classification
support vector machine
education technology
visual impairment application
datum processing method
recognition_of_cursive_English_handwritten_characters
image analysis
image preprocessing
character recognition challenge
machine print text vs handwritten text
pattern recognition
optical character recognition
image processing
document analysis
word segmentation
advanced technology
ocr technology
noise removal
skew normalization
document digitization
cursive handwriting
image quality
datum storage
handwritten text recognition
machine learn
word recognition
feature extraction
image acquisition
automatic number plate recognition
recognition accuracy
local benchmark database
multilingual ocr
binary image
neural network
ocr algorithm
datum mining
segmentation
computer science
handwriting recognition
A Detailed Analysis of Optical Character Recognition Technology
ocr history
ocr phase
research methodology
preprocesse technique
historical datum
research paper
A New Character Segmentation Approach for Off-Line Cursive
receipt imaging
handwriting sample
automate system
text extraction
literature review
english language processing
classification
postprocesse
grayscale conversion
segmentation error
information retrieval
character segmentation
captcha
performance analysis
computational science
morphological operation
banking application
healthcare application
conference proceeding
oversegmentation
ligature
algorithm development


Graph view
Labels
A Detailed Analysis of ...
A New Character Segm...
Recognition_of_cursive...
Thesis_Vault
datum processing method
recognition_of_cursive_English_handwritten_characters
image analysis
image preprocessing
image processing
image quality
image acquisition
binary image
A Detailed Analysis of Optical Character Recognition Technology
A New Character Segmentation Approach for Off-Line Cursive


Graph view
Labels
A Detailed Analysis of ...
A New Character Segm...
Recognition_of_cursive...
Thesis_Vault
Graph view
Recognition_of_cursive_English_handwritten_characters
neural network
A Detailed Analysis of Optical Character Recognition Technology
A New Character Segmentation Approach for Off-Line Cursive