



TANSZÉKVEZETŐ

SZAKDOLGOZAT FELADAT

Frey Dominik (AXHBUS)

BSc mérnökinformatikus hallgató részére

Deep learning alapú text mining magyar nyelvű adásvételi szerződések analizálására

Az emberi természetes nyelv gépi értelmezése és az úgynevezett “text mining”, már régóta a nagyvállalatok elengedhetetlen eszköze például felhasználói visszajelzések elemzésére vagy célzott hirdetések feladására. Fontos, hogy a nagy mennyiségű előzetes információval betanított mesterséges intelligencián alapuló algoritmusok, egy természetes nyelvű szöveget, annak kitisztítása után olyan módon dolgozzanak fel, hogy ezzel értékes információt tudjanak kinyerni, például statisztikákat vagy szöveg mögötti érzelmeket, benyomásokat.

A megvalósítandó rendszer feladata, hogy a magyar nyelvű alapadatokban megtalálja a felhasználó számára meghatározó fontosságú információkat, majd azokat – szükség esetén kiegészítő információkkal ellátva – csoportosítsa, tárolja és elérhetővé tegye.

A feladat megvalósítása a következő feladatok elkészítésére terjedjen ki:

1. Egy adásvételi szerződés szövegének értelmezése MI eszközök segítségével, valamint az algoritmus megértése és bemutatása.
2. Elegendő tanító szöveg gyűjtése / generálása az algoritmus betanítására.
3. Egy jól kezelhető felhasználói felület a szerződések beolvasására és a már beolvasott szerződések közötti böngészéshez.
4. A feldolgozott szerződések adatbázisba mentése.
5. A feladatok kettéválasztása egy kliens és szerver alkalmazásra, hogy a számításigényes feladatok a szerveren hajtsódjanak végre.
6. Részletes dokumentáció készítése.

Tanszéki konzulens: Max Gyula, adjunktus

Budapest, 2021. 09. 23.

Dr. Charaf Hassan
egyetemi tanár
tanszékvezető





Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Frey Dominik

TEXT MINING, MAGYAR NYELVŰ ADÁSVÉTELI SZERZŐDÉSEK ANALIZÁLÁSÁRA

Deep learning alapú eszközökkel

KONZULENS

Dr. Max Gyula

BUDAPEST, 2021

Tartalomjegyzék

Összefoglaló	6
Abstract.....	7
1 Bevezetés	8
1.1 A feladat értelmezése	8
1.2 A feladat indokoltsága	9
1.3 Tartalmi összefoglaló.....	9
2 Specifikáció.....	11
2.1 A rendszer architektúrája	11
2.2 Rendszerszintű követelmények.....	11
2.3 Kliens oldali követelmények	12
2.4 Szerver oldali követelmények.....	13
2.5 A keresett adatok	15
3 Előzmények, Felhasznált technológiák	17
3.1 Named Entity Recognition (NER)	17
3.1.1 PyTorch.....	18
3.1.2 Pandal	18
3.1.3 Hugging Face – transformers.....	18
3.1.4 Transformers architektúra.....	18
3.1.5 BERT	22
3.2 Asztali alkalmazás	23
3.2.1 Material Skin.....	24
3.2.2 IronOcr.....	24
3.3 Kommunikáció.....	24
3.3.1 Tcp-Socket	24
3.3.2 Super Simple Tcp.....	25
3.4 Adatbázis	25
3.4.1 System.Linq	26
4 Entitások kinyerése nyers szövegből	27
4.1 Modellben felhasznált adatok	27
4.1.1 Adatok begyűjtése.....	27
4.1.2 Vizuális analízis	28

4.1.3 Adatok előkészítése	31
4.2 NLP algoritmus.....	31
4.2.1 A modell finomhangolása.....	31
4.2.2 Eredmény kinyerése.....	33
4.3 Szerződés specifikus adatok	35
4.3.1 Adatok begyűjtése.....	35
4.3.2 Adatok formátuma	35
4.4 Regex alapú algoritmus	36
4.4.1 Attribútumok keresése formátum alapján.....	37
4.4.2 Validálás egyszerű kontextussal	38
4.4.3 Validálás összetett kontextussal.....	38
4.5 A két algoritmus összefonása.....	39
5 Az alkalmazások részletes elemzése	41
5.1 Kliens alkalmazás	41
5.1.1 Felépítése	41
5.1.2 Nézet interfész	42
5.1.3 Adatok megjelenítése.....	43
5.1.4 Képszedet.....	43
5.2 Szerveralkalmazás	45
5.2.1 Felépítése	46
5.2.2 Szerződések analizálása	47
5.2.3 Authentikáció és Authorizáció.....	50
5.2.4 Lekérdezések visszafejtése	51
5.2.5 Képszedet.....	52
5.3 Socket-API és kommunikáció	54
5.3.1 Socket-API.....	54
5.3.2 Definiált üzenetek	56
5.4 Adatbázis	57
6 Összegzés.....	59
6.1 Eredmények	59
6.2 Továbbfejlesztési lehetőségek	60
7 Irodalomjegyzék.....	61
8 Függelék.....	63
8.1 Fontosabb attribútumok	63

8.2 Az elkészült projekt	63
--------------------------------	----

HALLGATÓI NYILATKOZAT

Alulírott **Frey Dominik**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzé tegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 09.

.....
Frey Dominik

Összefoglaló

Az emberi természetes nyelv gépi értelmezése és az úgynevezett “text mining”, már régóta a nagyvállalatok elengedhetetlen eszköze például felhasználói visszajelzések elemzésére vagy célzott hirdetések feladására. Fontos, hogy a nagy mennyiségű előzetes információval vagy “corpus-szal” betanított mesterséges intelligencián alapuló algoritmusok egy természetes nyelvű szöveget, annak “megtisztítása” után olyan módon dolgozzanak fel, hogy ezzel értékes információt tudjanak kinyerni belőle, például statisztikákat vagy szöveg mögötti érzelmeket, benyomásokat.

A megvalósítandó rendszer feladata, hogy a magyar nyelvű alapadatokból (különböző formátumú adásvételi szerződésekből) kikeresse a felhasználó számára meghatározó fontosságú információkat, majd azokat – szükség esetén kiegészítő információkkal ellátva – csoportosítsa és tárolja. Ehhez segítséget nyújt majd az adásvételi szerződések valamilyen szinten kötött szerkezete és a Deep Learning alapú NLP (Natural Language Processing) technológiák. A tárolást követően biztosítani kell még a kinyert adatok közti böngészést egy kliens felületen, lehetőleg a feldolgozást végző alkalmazástól független módon az ehhez szükséges kommunikáció megvalósításával, kliens oldal és szerver oldal között. A köré épülő komplett szoftveren kívül alapvető cél továbbá a Python-ban vagy Java-ban implementált NLP technológiák algoritmusának megismerése és felhasználása a magyar nyelvű nyers szövegek értelmezésére.

Abstract

Machine interpretation of human natural language, and the so-called “text mining,” has long been an essential tool for large corporations, for example, to analyze user feedback or post targeted advertisements. It is important that algorithms based on artificial intelligence taught with a large amount of prior information or “corpus” process a natural language text after it is “cleansed” in such a way that it can extract valuable information from it, such as statistics or emotions behind the text and impressions.

The task of the system to be implemented is to retrieve the information of decisive importance for the user from the basic data in Hungarian (sales contracts of various formats), and then to group and store them - with additional information if necessary. This will be aided by a level-based structure of sales contracts and Deep Learning-based NLP (Natural Language Processing) technologies. After storage, browsing between the extracted data on a client interface must be ensured, preferably in a manner independent of the application performing the processing, by implementing the necessary communication between client side and server side. In addition to the complete software built around it, the basic goal is to get to know the algorithm of NLP technologies implemented in Python or Java and to use it to interpret raw texts in Hungarian.

1 Bevezetés

Ebben a fejezetben olvasható, hogy személy szerint hogyan értelmeztem az adott feladatot és, hogy a megoldás közben mi motivált a fejlesztésre, egyben mi indokolja egy ilyen projekt létjogosultságát. A fejezet végén mondatokba foglalva bemutatom a dolgozat tartalmi elemeit.

1.1 A feladat értelmezése

Ezen dolgozat keretein belül a feladat egy rendszer megvalósítása, amelynek elsődleges funkciója magyar nyelvű, adásvételi szerződések szövegének az értelmezése. Az értelmezésnek úgy kell történnie, hogy a végeredmény a szerződésekben megtalálható, általam definiált adatok és azok kapcsolatának egy program által kezelhető absztrakciója legyen. Ehhez elvárt egy vagy több gépi tanuláson alapuló algoritmus helyes kiválasztása és alkalmazása, valamint ezen algoritmusok működésének a megértése és adott szintű kifejtése, mint technológiai háttér. Az algoritmus tanításához egy nagy méretű tanító adathalmazt kell begyűjtenem.

A rendszer megvalósításához ki kell választanom egy megfelelő programozási nyelvet vagy nyelveket, olyat, amely a feladat elvégzése szempontjából indokolt.

A szövegértelmező funkción kívül, a rendszernek képesnek kell lennie a kinyert adatszerkezet permanens és hatékony tárolására egy általam kiválasztott adatbázis szerveren. Az adatbázis tartalmának eléréséhez egy kliens felület kialakítása is feladat, ahol a felhasználó kényelmesen el tudja végezni a támogatott CRUD műveleteket. Ez jelenti azt, hogy az elmentett adatok a felhasználói felületen megjeleníthetők és szűrhetők, valamint egy szerződés beolvasásával a felhasználó módosíthatja az adatbázis tartalmát.

A rendszert két külön komponensre kell bontanom, egy szerver és egy kliens alkalmazásra, hogy a hosszú számítási feladatok ne a kliens oldalt terheljék. Ebből adódik, hogy a két fél közötti kommunikáció kialakítása is a feladat része, amihez egy a célnak megfelelő kommunikációs technológiát kell kiválasztanom. Továbbá elvárás több felhasználó kiszolgálása egy időben anélkül, hogy az egyszerre érkező kérések konfliktust okoznának. A felhasználókat egy adatbázisban megfelelően nyilván kell tartani és mind kliens, mind szerver oldalon azonosítani a kapcsolat létrejöttékor. Ezen

kívül a felhasználóknak különböző jogköröket kell kiosztani és az alapján engedélyezni egyes műveletek elérését szintén mindkét oldalon.

1.2 A feladat indokltsága

A polgári jogban az adásvétel, illetőleg az adásvételi szerződés az egyik leggyakrabban előforduló jogviszony, illetve szerződéstípus. Ingatlan adásvételi szerződés esetén a szerződést ügyvédnek, jogtanácsosnak vagy közjegyzőnek kell ellenjegyeznie [1].

Manapság már a legtöbb szakterületen jellemző, hogy a repetitív, ember számára sok idő és nagy erőforrásigényű feladatokat automatizálják annak érdekében, hogy a gépek segítségével az emberi erőforrások minél hatékonyabban lehessenek kihasználhatók. Habár ezekre a feladatokra sok esetben nem triviális algoritmust írni, az információs technológia már eljutott arra a szintre, hogy okos megoldásokkal és a mesterséges intelligencia jelenleg fellelhető eszköztárának segítségével, egyre komplexebb feladatokat tudjunk kivenni az emberi kéz alól. Ez alól nem kivétel a természetes nyelvű szöveg értelmezése és egy átfogó kép kialakítása annak tartalmáról.

Hasonló feladat, amikor különböző jogi esetekben adásvételi szerződéseket kell előhívni és annak tárgyait, résztvevőit, akár több szerződésen keresztül visszakövetni és ellenőrizni. Ha lenne egy olyan digitális adatbázis, ahová a szerződésekben fellelhető főbb információk egy kattintással felvihetők, majd bármely attribútum álltali keresés során lehívhatók, az lényegesen felgyorsítaná a munkafolyamatot. A munkámat ez a lehetőség motiválta és a célom, egy olyan asztali alkalmazás és adatbázis létrehozása, ami mindezt lehetővé tenné.

1.3 Tartalmi összefoglaló

A dolgozat a feladat részletes elemzésével kezdődik [Specifikáció](#) címszó alatt, ahol meghatározásra kerül a rendszertől elvárt architektúra, a rendszer egyes komponenseinek funkcionális követelményei és use-case-ei. Ezen kívül specifikálom a program által a nyers szövegben keresendő entitásokat és a hozzájuk tartozó attribútumokat.

Az ezt követő [Előzmények és felhasznált technológiák](#) fejezetben a megvalósítás során felhasznált technológiákból és segédkönyvtárakból sorolom fel az általam legfontosabbnak ítélteteket és kellő részletességgel bemutatom őket és az elveket, amik

alapján működnek. Mindezt betöltött funkciójuk szerint csoportosítva, így szó lesz a nyelvfeldolgozást, a kommunikációt és az adatbáziskezelést megvalósító technológiákról és néhány, az asztali alkalmazással kapcsolatos könyvtárról és környezetről.

A következő két fejezet a rendszer megvalósításáról fog szólni.

Az első, az [Entitások kinyerése nyers szövegből](#), ami a nyelvfeldolgozás algoritmusait mutatja be részletesen. Először a szükséges adathalmazok begyűjtésének és formázásának lépéseit írom le, majd a modell előkészítését és felhasználásának módjára térek ki. Két fajta algoritmust mutatok be, az egyik a gépi tanuláson alapuló, a másik pedig a saját magam által felépített, reguláris kifejezésekkel operáló megközelítés. A fejezet végén a két algoritmus együttes felhasználásának módját vezetem le.

A második, [Az alkalmazások részletes elemzése](#) fejezetben az alkalmazás komponenseinek egyes funkcióit és felépítését részletezem. Elsőként a kliens, majd a szerver, végül a kommunikáció és az adatbáziskezelés részleteit mutatom be.

Végül az [Összegzésben](#) összefoglalom az elvégzett munkát és a tapasztalataimat, számba veszem az elért eredményeket, amiket a munka ezen fázisán sikerült elérnem és velük együtt a hiányosságokat is. Ebből kiindulva felsorolom a lehetséges fejlesztéseket a jövőre nézve.

Az [Irodalomjegyzék](#) után, a dolgozat végén egy [Függelék](#) található, amiben felsorolom az összes olyan attribútumot, kivétel nélkül, amit egy szövegben keres a program, valamint az elkészült alkalmazásról szúrok be elérési útvonalat, hogy bármely olvasó letölthesse és megtekinthesse azt.

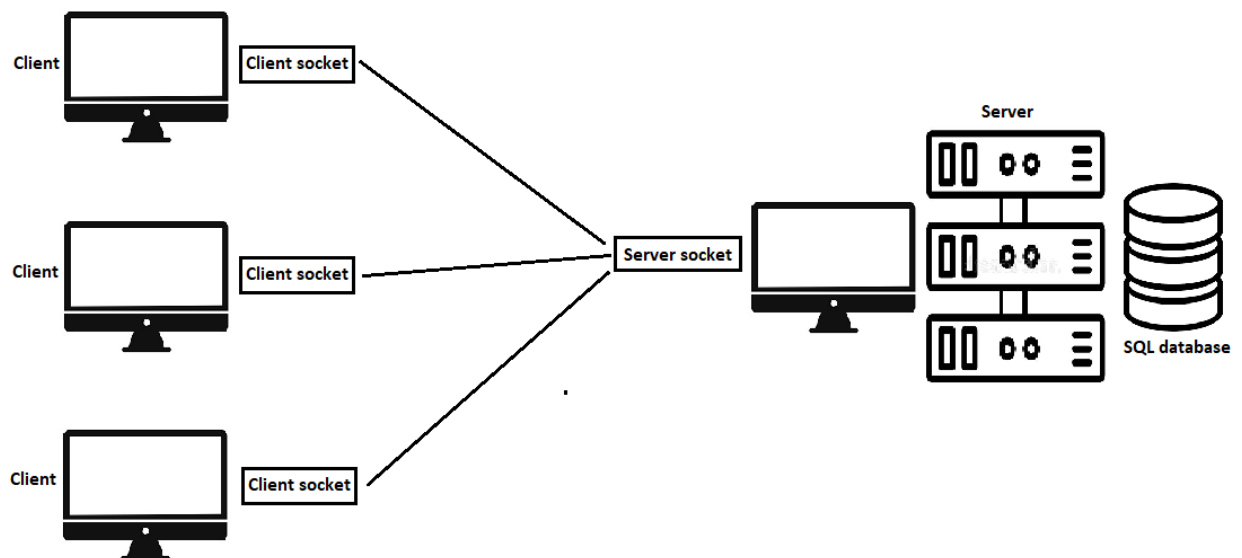
2 Specifikáció

Ebben a fejezetben magas szinten bemutatásra kerül, hogy mit várunk el a rendszertől, mint olyan és annak különböző részegységeitől. Itt a felhasznált technológiák még nem kerülnek részletezésre, ellenben a feladat pontosítása és részletes elemzése kerül bemutatásra.

2.1 A rendszer architektúrája

A rendszer kliens szerver architektúrát követ, ahol mind a kliens, mind a szerver asztali alkalmazások. A szerver kapcsolatban áll az adatbázissal a kinyert adatok permanens tárolása céljából, és egy API-n keresztül fogadja a kliens kapcsolatokat. Ez látható az **2.1. ábrán**.

Socketen keresztül hatékonyan küldhetők nagy méretű adatsomagok, valamint a szerver képes tájékoztatni a klienseket, ha változás történt a szerver oldalon, ezért döntöttem úgy, hogy a kommunikációt ilyen alapon kell majd megvalósítanom.



2.1. ábra – Az elvárt rendszer architektúra

2.2 Rendszerszintű követelmények

A rendszer alapvető célja, hogy a bemeneten kapott adásvételi szerződéseket feldolgozza, a kinyert adatokat tárolja és adott esetben a felhasználónak biztosítsa. Ezek közül a feldolgozás igényli a legtöbb számítási időt, a komplex számítások miatt nem is

keveset, ezért felmerül a kérdés, hogy célszerű-e a felhasználói eszközökön végezni ezt a műveletet. Mivel a feldolgozásnak viszonylag gyorsan kell végbe mennie, de nincs igény valós idejű válaszra, amellet döntöttem, hogy maga a kliens csak a szövegek beolvasásával és továbbításával járuljon hozzá a szöveg analizálásához, minden további számítás szerveroldalon történjen.

Ez több szempontból is előnyös:

- Egy gyors kliens felületet kapunk, nem kell a számításokra várakoznia a felhasználónak. Feldolgozás közben a kliensen más feladatok is elvégezhetőek.
- Esetleges internethiba vagy a program bezárását követően sem szakadnak meg a műveletek, mivel a szöveg elküldése után a szerver már gondoskodik a végrehajtásról.
- Egyszerűbb, átláthatóbb kliens alkalmazást kapunk.

Persze néhány hátrulütővel is kell számolnunk:

- Előfordulhat, hogy a szerveren felhalmozódnak a feladatok és jelentősen csökken a válaszideje. Ez nem akkora probléma, ha közben a bejövő lekérdezéseket ki tudja szolgálni. Ennek biztosítása is alapvető követelmény.
- A szerver sokrétűsége miatt komplexitása megnövekedik, ezt jó architektúrával és helyes programozással menedzselni kell.

2.3 Kliens oldali követelmények

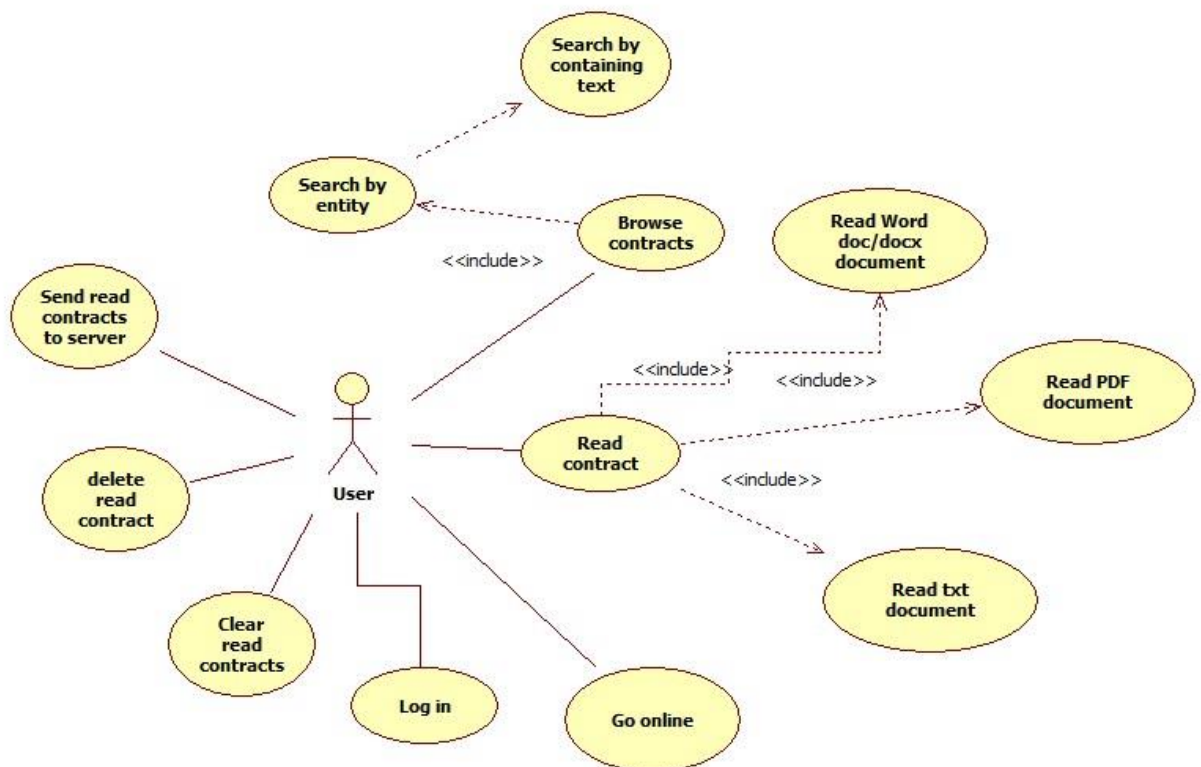
A kliens egy asztali alkalmazás, jól átlátható grafikus felülettel rendelkezik. Felelős az adásvételi szerződések beolvasásáért, továbbításáért a szerver és a szervertől lekérdezett adatok megjelenítéséért.

Indításkor a kliens automatikusan kapcsolódik a szerveralkalmazáshoz, de lehetőség van manuálisan „online” módba váltani, ha a felhasználói igény megköveteli.

Csatlakozás után a felhasználónak be kell jelentkeznie felhasználónév és jelszó megadásával, csak ezután jogosult a legtöbb funkció igénybevételére.

A szerződések beolvasását .DOC, .DOCX, .PDF, .TXT formátumokra kell tudnia támogatnia a kliensnek. A már beolvasott szerződéseket ezután lehetőség van továbbítani a szerverre feldolgozásra. A már beolvasott szerződéseket küldés előtt lehet törölni vagy újakat beolvasni tetszés szerint.

A szerződések megtekintéséhez egy külön ablak jelenik meg, amely kényelmes böngészést biztosít. A felhasználó kiválaszthatja mely attribútumokat szeretné látni az adatbázisból majd kérést intéz a szerverhez. A kapott adatokat kilistázza, amelyek között tartalmazott szöveg alapján tud szűrni a felhasználó. A felhasználó által elvégezhető feladatokat hivatott demonstrálni a **2.2. ábrán** látható use-case diagramm.



2.2. ábra - A kliens alkalmazás use-case diagrammja, MagicDraw segítségével készítve

2.4 Szerver oldali követelmények

Általánosságban a szerver felelős azért, hogy a kliensek kérései ki legyenek szolgálva, tehát alapvető feladata a kapcsolatok fogadása és menedzselése, valamint a bejövő kérések értelmezése és a megfelelő válasz küldése egy API-n keresztül. A szervernek ezen felül gondoskodnia kell arról, hogy illetéktelen felhasználók ne férjenek hozzá a tárolt adatokhoz. Ennek érdekében csak olyan kliensek kéréseit szolgálja ki, amelyek előzetesen azonosították magukat és elég magas jogkörrel rendelkeznek.

Feladatok, amit a szerver ellát a klienssel való interakció keretein belül:

- Klienssel való kapcsolatok kiépítése és bontása.
- Felhasználók autentikációja és autorizációja.

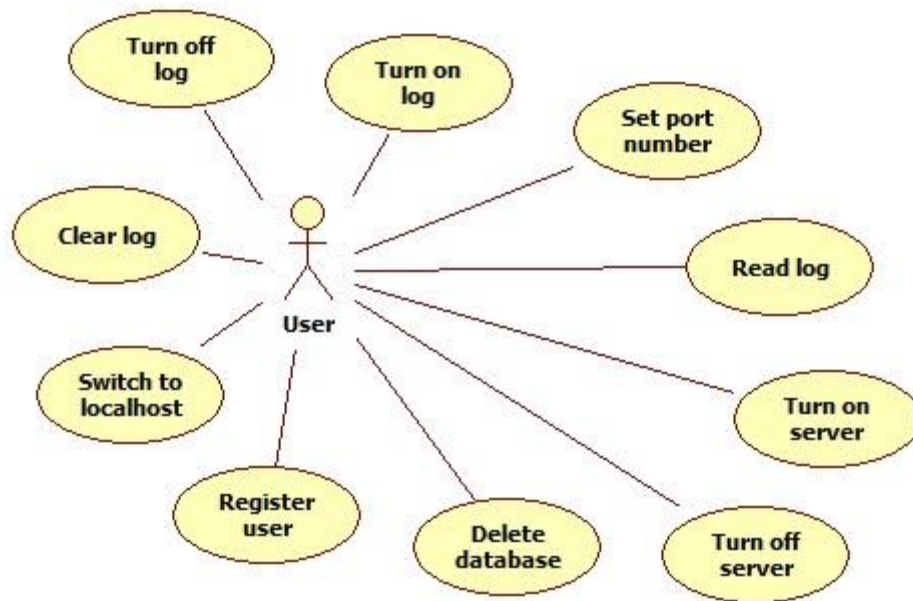
- Szerződés-feldolgozási kérések fogadása és végrehajtása.
- A feldolgozás állapotáról informatív üzenetek küldése.
- Adatlekérdezési kérések fogadása és kiszolgálása.
- Az adatbázis módosulása esetén minden „online” kliens értesítése.

Egy feldolgozási kérés érkezésekor a szervernek el kell végeznie a szöveg analizálását, az adatok kinyerését, validálását, utólagos formázását és csoportosítását, majd az eredményt el kell mentenie az adatbázisba, mindezt a lehető leghatékonyabban idő és pontosság szempontjából. Az adatbázisnak nagy mennyiségű adat tárolása esetén is hatékonyan lekérdezhetőnek kell lennie.

A szerver maga is egy asztali alkalmazás, egyes műveletek könnyebb elvégzése érdekében kap grafikus felületet, ahol a felhasználó képes:

- Részletes logüzeneteket olvasni a futásról.
- Elindítani, illetve lekapcsolni a szervert.
- Beállítani a portot, ahol a szerver hallgatózik.
- Új felhasználót regisztrálni felhasználónév, jelszó és jogosultsági kör megadásával.
- Törölni az adatbázist.

A felhasználó által elvégezhető feladatokat hivatott demonstrálni a **2.3. ábrán** látható use-case diagramm.



2.3. ábra - A szerveralkalmazás use-case diagrammja, MagicDraw segítségével készítve

2.5 A keresett adatok

Azt, hogy egy adásvételi szerződésben melyek a számunkra releváns információk, elsőre nehéz megmondani, hiszen egy ilyen szerződésnek lehet akár minden egyes szava fontos. Szerencsére az alkalmazás célja nem a logikai kikapuk feltárása. Olyan adatokat keresünk, amelyek sokszor elő fordulnak egyes fajta szerződésekben. Most itt egyelőre csak az ingatlan és gépjármű adásvételi szerződésekre fókuszálunk és az ezekben keresett adatokat specifikáljuk. Ezen adatoknak jól definiálnak és számuknak végesnek kell lenniük, mivel mielőtt megtalálunk egy adatot tudnunk kell, hogy mit keresünk. Mivel a kutatás során számos ilyen attribútumot gyűjtöttem ki, most csak említés szerűen, a teljesség igénye nélkül sorolok fel néhányat, de megjegyzendő, hogy a lista bővíthetősége is a követelmények közé tartozik.

- A szerződések általános információi:
 - keltezés, az adásvételi szerződés típusa (pl. ingatlan adásvételi szerződés), ellenjegyző iroda.
- A szerződés résztvevői:
 - Az eladók és vevők, ezen belül, ha az illető személy akkor, személyi azonosító, adóazonosító, születési hely és dátum, lakcím, elő- és utónév, anyja neve, bankszámlaszám.

- Ha az illető szervezet akkor, a szervezet neve, típusa (pl. kft.), székhelye, adószáma, statisztikai azonosítója, cégjegyzék száma, számlaszáma.
- A szerződés tárgyai:
 - Ha a szerződés tárgya egy ingatlan, akkor: az ingatlan típusa (pl. szántó), címe, területe, a közös tulajdoni hányad, szobák száma (pl. lakásnál), helyrajzi száma, belterületen fekszik-e és a földhivatal neve.
 - Ha a szerződés tárgya egy gépjármű: a gépjármű rendszáma, motorszáma, alvázszáma, típusa, forgalmi száma, extra felszerelései, első forgalomba helyezésének éve, kilométer száma és a tény, hogy használt-e.
- A szerződés teljesítésének feltételei:
 - A vételár, áfa, részteljesítések száma, a határidő és a tény, hogy banki átutalással fizetnek

Mindezek a [Függelékben](#) kivétel nélkül felsorolásra kerülnek.

3 Előzmények, Felhasznált technológiák

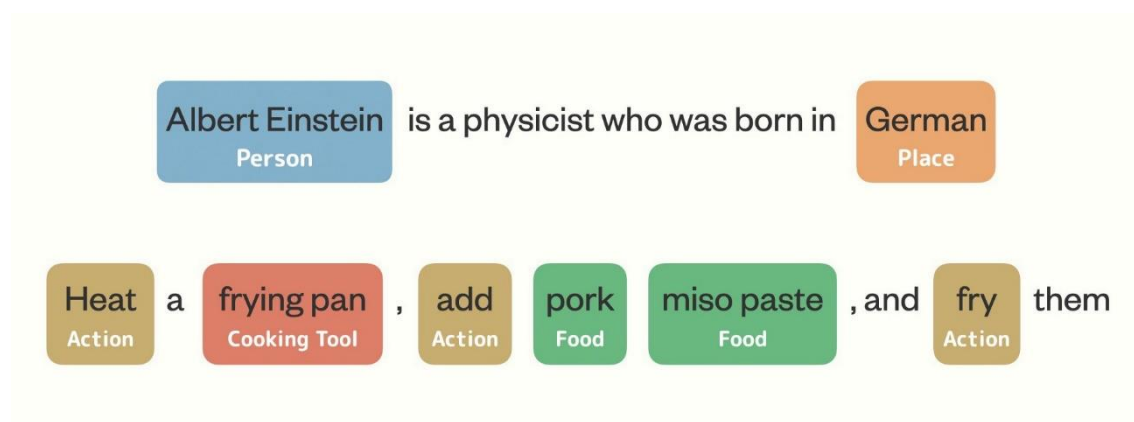
A témával kapcsolatban viszonylag sok tapasztalatot és forrást halmoztam fel, mivel a témalaboratórium és önálló laboratórium keretein belül is hasonló feladaton dolgoztam. Ugyanakkor, ezen dolgozat egy kicsivel több kutatást igényelt, számos új technológiának néztem utána és használtam a rendszer megvalósításához. Ezek közül a munkám szempontjából fontosnak vagy érdekesnek tartottakat szeretném ismertetni, azokban ésszerű szintekig elmélyülve.

3.1 Named Entity Recognition (NER)

Az alkalmazás azon része, amely a nyers szövegből a számunkra érdekes adatok kiolvasásáért felelős Named Entity Recognition-ra (más szavakkal entitás identifikációra) épít. Erre a következőkben csak NER-ként fogok hivatkozni.

A NER a Natural Language Processing (NLP) egy módszere, amely egy struktúrátlan szövegben hivatott megtalálni és előre megnevezett kategóriákba sorolni a megemlített entitásokat, legyen az személy, szervezet, helyszín és még sok más.

A gépi tanulás alapú NER megvalósítását teljes egészében Python (3.8.8) [2] programozási nyelvben írtam, mivel a nyelvben megvalósított NLP könyvtárak tárháza igen gazdag. Ebben a fejezetben az általam használt modulok kerülnek bemutatásra. A NER működésének lényegét az **3.1. ábrán** lehet megtekinteni, ahol az látszik, hogy a szöveg egyes entitásai különféle klasszba lettek sorolva a klasszifikációt elvégző program által.



3.1. ábra - Példa a NER működésének demonstrálására, forrás: [20]

3.1.1 PyTorch

PyTorch [3] egy ingyenes mesterséges intelligencia keretrendszer Pythonra és annak torch moduljára alapozva, amit a Facebook mesterséges intelligencia kutató csapata fejlesztett ki. Támogatja a tenzorokon való műveletvégzést a GPU igénybevételével. Jelenleg a Pytorch a legelterjedtebb Deep Learning könyvtár a mesterséges intelligenciát kutatók körében.

3.1.2 Pandas

Pandas [6] egy 2008-ban kiadott, ingyenes adatkezelő függvénykönyvtár a Python felett. A könyvtár adatok analizálására és manipulálására használható. Leglényegesebb funkciója az úgynevezett DataFrame, amely egy kétdimenziós, méretezhető, táblázatos adatstruktúra. Jól használható különböző formátumú adatok importálására, mint például XML, JSON, CSV, Excel. A könyvtárat arra használtam, hogy CSV adathalmazokat tölthessek be, felcímkézhessem az oszlopaikat és kiszűrhessem a számomra fontos részhalmazt és megfigyelhessem, majd használhassam azt modell betanításra.

3.1.3 Hugging Face – transformers

A Hugging Face [4] egy mesterséges intelligencián alapuló chat-bot alkalmazásokat fejlesztő vállalat, amelyet 2016-ban alapítottak. Git-alapú tárhelyeik tárolóként működnek, és a projekt összes fájlját tartalmazhatják, Github-szerű funkciókat kínálnak. Rengeteg más fejlesztő által közzétett, ingyenes, előre tanított modell és adathalmaz lelhető fel rajta.

A cég által fejlesztett transformers [5] könyvtár egy jól felszerelt API-t tesz közzé, a rengeteg Deep Learning architektúra használatára a legelterjedtebb NLP problémák megoldásához, mint például NER, Q&A, szöveg klasszifikáció. A transformers könyvtár több mint 30 előre tanított modellt tartalmaz, némelyik akár 100 nyelven is használható.

3.1.4 Transformers architektúra

A Transformers [17] a Google és a Torontói egyetem mesterséges intelligencia kutatói által 2017-ben kifejlesztett úttörő NLP architektúra, amelynek célja a „sequence-to-sequence”, „sequence-to-vector” és „vector-to-sequence” problémák megoldása. Ez azt jelenti, hogy bemenetként szavak sorozatát vagy vektort vár, kimenetként pedig szintén szavak sorozatát vagy vektort ad vissza. Jó példa egy „sequence-to-sequence”

problémára egy magyar-angol fordító program, amely egy magyar mondatot vár a bemenetén, a kimeneten pedig a megfelelő angol mondatot kapjuk.

A Transformers előtt ilyen célokra RNN (Recurrent Neural Network) és LSTM (Long Short-Term Memory Network) technológiákat használtak, de ezeknek nehezebb volt a hosszabb szövegek kezelése, valamint általánosan lassúak voltak. A Transformers újítása az előzőekhez képest, hogy egy mondat szavait egyszerre olvassa be, nem pedig sorrendben, azaz a szavakat a sorrendiségtől függetlenül értelmezi. Ez azzal az előnnyel jár, hogy jobban ki tudja használni a GPU párhuzamos végrehajtását. Emellett, így az eljárás nem használ se rekurziót, se konvolúciót, hanem helyette a későbbiekben tárgyalt „positional encoding” módszerével kezeli a szavak mondatbeli pozícióját.

A következőkben ésszerű részletességgel bemutatom a Transformers architektúráját egységeire bontva, az eredeti tanulmány alapján:

Az architektúra két fő elemből áll egy Encoderből és egy Decoderből. Ezen kívül a bemenetek egy előfeldolgozáson mennek keresztül.

Input Embedding: Mivel a számítógép nem tud tényleges szavakon műveleteket végezni, feldolgozás előtt a bemenetet le kell képezni egy vektor térbe. Ez az úgynevezett „Input Embedding”, ami minden szóhoz egy vektort rendel úgy, hogy az egymással, jelentésben hasonló kategóriába tartozó szavak a térben minél közelebb kerüljenek egymáshoz.

Positional Encoding: Egy-egy mondatban ugyanazon szavak más jelentéssel bírhatnak mondatbeli pozíciójuk függvényében. Ezáltal a hozzájuk rendelt vektor nem lehet ugyanaz. A „Positional Encoder” a szavakból vektort képez pozíciójuk alapján és hozzáadja a már rendelkezésre álló „Input Embedding” kimenetéhez, ezáltal a szó kontextust kap. Az eredeti tanulmányban ez a vektor így áll elő:

$$PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$$

Ahol:

pos: a szó pozíciója, i: a vektor aktuális sora, d_{model} : a model dimenziója

Multi-Head Attention: Ebben a kontextusban az „Attention” azt jelenti, hogy egy bemenetként kapott mondatban mely szóra vagy szavakra érdemes jobban fókuszálni. Ennek fényében, minden i-edik szóra a mondatban generálódik egy úgynevezett

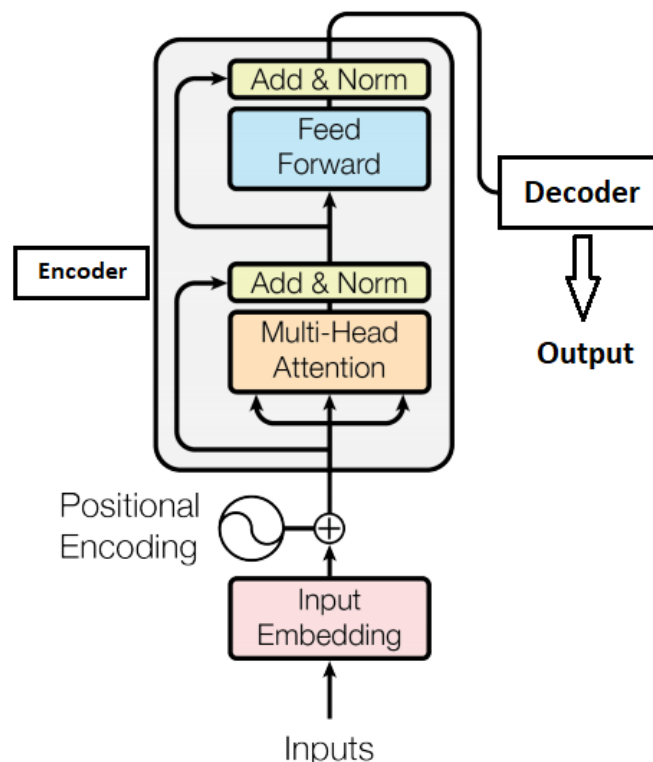
„Attention Vector”, ami a szavak számával egyenlő dimenziójú vektor és azt jelképezi, hogy egyes szavak mennyire relevánsak az i-edik szó tekintetében.

Feed Forward: Valójában a modell 6 darab Decodert és 6 darab Encodert tartalmaz, amik a „Feed Forward” neurális háló által vannak kapcsolatban.

Encoder:

Az Encoder bemenetére kerül az a szó, mondat vagy vektor, amire a feladat jellegétől függően valamiféle választ szeretnénk kapni. Magyar-angol fordítás esetén például a magyar mondat.

Az előfeldolgozott bemenet az Encoderbe kerül vektorok formájában hozzárendelt jelentés és kontextus alakban. Ez után a „Multi-Head Attention” blokkban létrejön minden szóhoz egy „Attention Vector”, amiket párhuzamosan továbbítunk a „Feed Forward” segítségével a Decoderbe vagy egy következő Encoderbe. Eközben minden réteg kimenetén még további normalizációs eljárások kerülnének végrehajtásra. A Transformers Encoder részmodelljének architektúráját a **3.2.ábra** demonstrálja.



3.2. ábra - Transformers Encoder architektúra, forrás: [17]

Decoder:

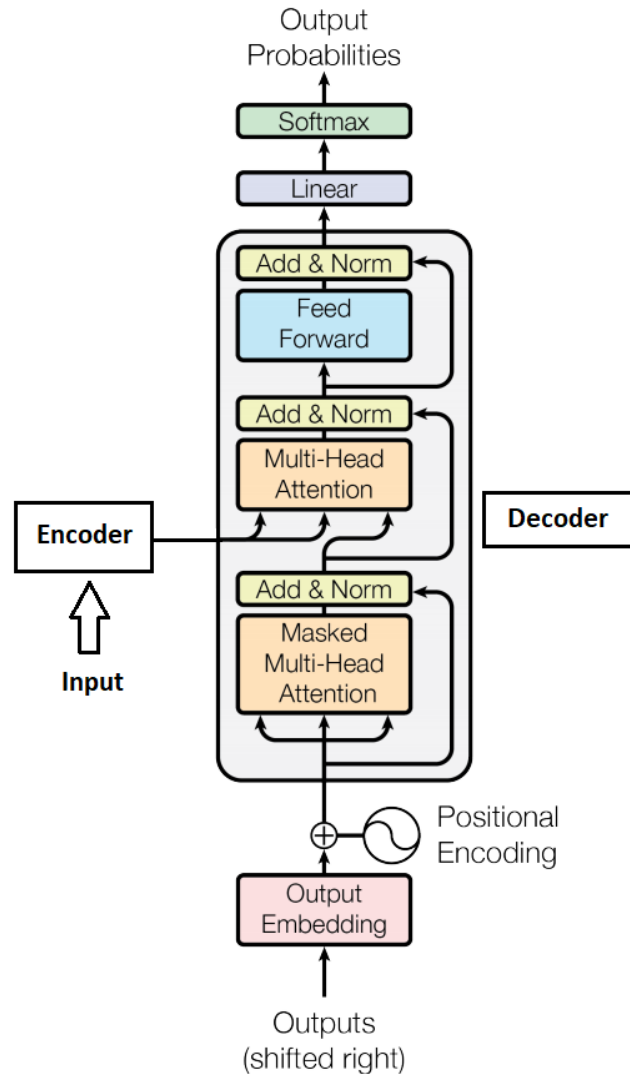
Az Decoder bemenetére kerül az a szó, mondat vagy vektor, ami a feladat jellegétől függően valamit megválaszol. Magyar-angol fordítás esetén például az angol mondat.

A Decoder esetén az elvárt kimenet az eddigiekhez hasonló módon kerül előfeldolgozásra, majd kerül be a Decoderbe, egy „Masked Multi-Head Attention” rétegbe. Ez annyiban más, mint az eredeti „Multi-Head Attention”, hogy a kimeneti „Attention Vector”-okban az i -edik szó után álló szavakhoz tartozó dimenziók értékei 0-val lesznek elmaszkolva. Ez azt jelenti, hogy csak az i -edik szó és az az előtt álló szavak vannak egyedül fókuszban. Ez azért fontos, mert a tanulás során csak a már előállított kimenetekből vonható le következtetés, a jövőben generált kimenetek még nem ismertek.

A következő réteg egy újabb „Multi-Head Attention” blokk. Itt történik az Encoder és a Decoder „Attention Vektor”-ainak páronkénti „összefonása”, ahol kimenetként újabb „Attention Vector”-okat kapunk.

Ezt követően a vektorok a „Feed-Forward” rétegen keresztül vagy újabb Decoderekbe kerülnek vagy egy úgynevezett „Linear Layer”-be, amely maga is egy „Feed-Forward” réteg. Itt vektor dimenzió kiterjesztés történik, minden vektor kiterjed a már ismert szószeret méretével egyenlő dimenzióra. A kiterjesztett vektor minden, egy-egy szót jelképező eleméhez egy érték rendelődik.

Utolsó lépésként a „Softmax” rétegben a kapott vektor minden elemének értékéből pedig egy valószínűség generálódik, amelyből a legnagyobb valószínűséghez rendelt szó lesz kiválasztva, mint eredmény. A Transformers Encoder részmodelljének architektúráját a **3.3.ábra** demonstrálja.



3.3. ábra - Transformers Decoder architektúra, forrás: [17]

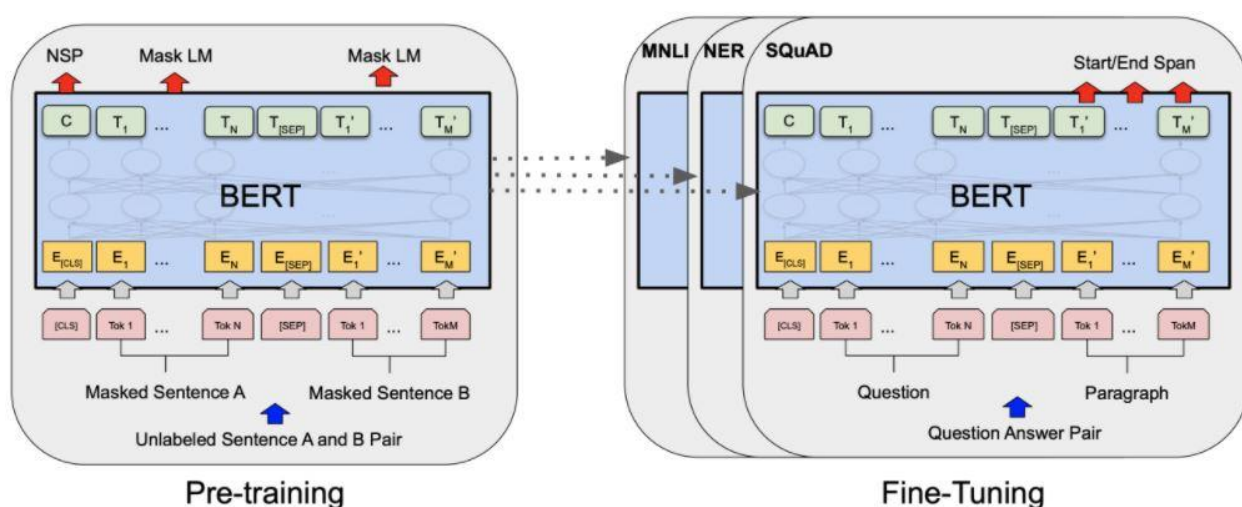
3.1.5 BERT

A BERT (Bidirectional Encoder Representations from Transformers) [13] egy friss tanulmány, amelyet a Google mesterséges intelligencia kutatói tettek közzé 2019-ben. Manapság az NLP feladatok megoldására a legkorszerűbb technológia.

A BERT újítása, hogy a [Transformers](#) modell bidirectional (kétirányú) tanítását alkalmazza. A kétirányúságnál pontosabb megfogalmazás, hogy irány nélküli mivel, a szósortozatot nem balról jobbra, sem jobbról balra, hanem egyszerre vizsgálja.

A Transformer két különálló mechanizmust tartalmaz: egy Encodert, amely beolvassa a szöveg bemenetet és egy Decodert, amely becslést készít a feladat megoldásáról. Mivel a BERT célja egy nyelvi modell létrehozása, csak az Encoder mechanizmusra van szükség.

Egy BERT modell használata előtt két lépést kell végrehajtani. Az egyik a modell betanítása nagy mennyiségű felcímkézett szöveg alapján, például Wikipédiáról. A második lépés a „Fine-Tuning”, azaz finomhangolása modellnek. Ebben a lépésben betöltjük a modellt az előre tanított paraméterekkel, majd egy felcímkézett speciális témakörre összeállított corpus segítségével egy további Transformer réteggel finomhangoljuk a már meglévő összes paramétert. A **3.4. ábrán** az látszik, ahogy a fentebb tárgyalt tanítást először felcímkézett szöveggel végezzük el, majd ugyan ezt az adott feladatra kialakított felcímkézett szöveggel, például egy kérdés megválaszoló modellt kérdés – válasz párokkal finomhangolunk.



3.4. ábra - BERT tanítás és finomhangolás, forrás: [13]

Szerencsére a [Hugging Face](https://huggingface.co/) által rengeteg előre tanított modell elérhető, akár több nyelven is, például a „bert-base-multilingual-cased” [18] elnevezésű modell, ami 12 Transformer rétegből áll, 174M paramétert tartalmaz és 104 természetes nyelvet támogat. Így az egyetlen lépés, amit el kell végezni az egy előre tanított BERT modell finomhangolása NER specifikus feladatra.

3.2 Asztali alkalmazás

Mind a szerver mind a kliens szoftver C# .NET keretrendszerben íródtak. A C# nyelv 4.0-as verzióját és a .NET keretrendszer 4.8-as verzióját használtam a megvalósításhoz. A két szoftver Windows Forms alkalmazások és Microsoft Visual Studio fejlesztői környezet segítségével fejlesztve. A C# programozási nyelv kényelmes, jól átlátható, objektum orientált programozást tesz lehetővé, ami egy robusztusabb

alkalmazás esetén fő szempont. Emellett rengeteg letölthető segédkönyvtár érhető el, amik közül az alábbiakat használtam:

3.2.1 Material Skin

A Windows Forms könyvtár felhasználói felület elemeinek viszonylag idejétmúlt megjelenését váltottam fel a Material Skin [7] könyvtár minimalista, de mutatósabb elemeivel egy felhasználóbarát interfész megvalósításának érdekében.

3.2.2 IronOcr

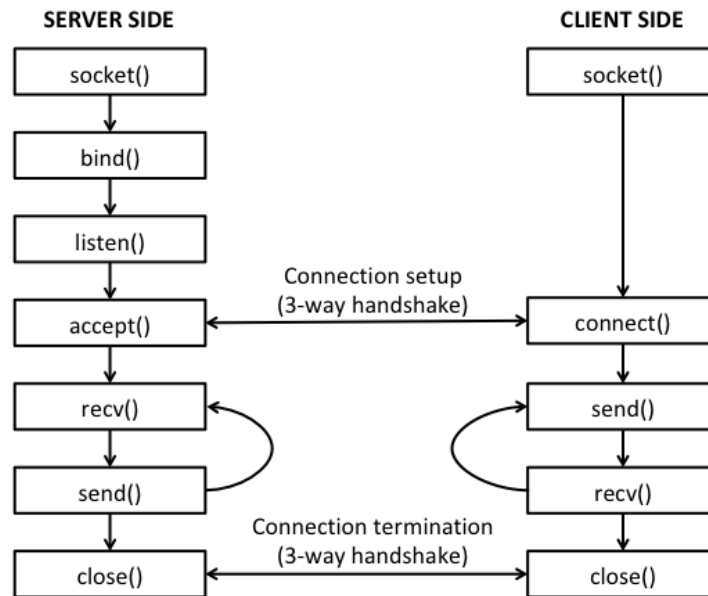
A PDF állományok beolvasására az IronOcr [9] könyvtárat használtam. Az IronOcr egy egyszerűen használható, Tesseract [10] motor felé épülő, karakter felismerő eszköz. PDF mellett képekről történő szöveg olvasására is használják és akár 99.8%-os pontosság is elérhető vele. Azért esett rá a választásom, mert mindemellett támogatja a magyar abc karaktereit és gyorsaság / pontosság orientáltan konfigurálható.

3.3 Kommunikáció

3.3.1 Tcp-Socket

Egy socket egy operációs rendszer absztrakció, ami egy kapcsolatot reprezentál. A Tcp-Socket lehetővé teszi az alkalmazások közötti kommunikációt, Tcp protokollt használva socketen keresztül.

A **3.5. ábra** ezt a kommunikációt vezeti le. A szerver először létrehoz egy új socketet a `socket()` hívással, majd a `bind()` hívással hozzákapcsolja egyik IP címét és portját, amin majd a későbbiekben a `listen()` hívással aktiválja a hallgatózást. Amikor egy kliens kapcsolatot kezdeményez és végbemegy a Tcp protokoll által előírt 3 utas kézfogás, kialakul a kapcsolat és mindkét fél a `send()` és `recv()` hívásokkal tud bájt sorozatokat küldeni vagy fogadni. A `close()` hívással lehet lebontani a Tcp kapcsolatot.



3.5. ábra - Tcp-Socket kommunikáció, forrás: [19]

Ezt a technológiát használja a hálózati kommunikációk nagyrésze, köré épült API-kkal vagy egyszerű üzenet küldésekkel, mert rugalmas és hatékony, valamint kis hálózati forgalmat generál.

A REST alapú kommunikációval ellentétben Socket API használatával a szerver nyilván tarthatja a klienseket és kezdeményezheti a kommunikációt, ha arra van szükség, például az adatbázis megváltozása esetén. Ennek a kihasználása érdekében döntöttem a Socket technológia mellett.

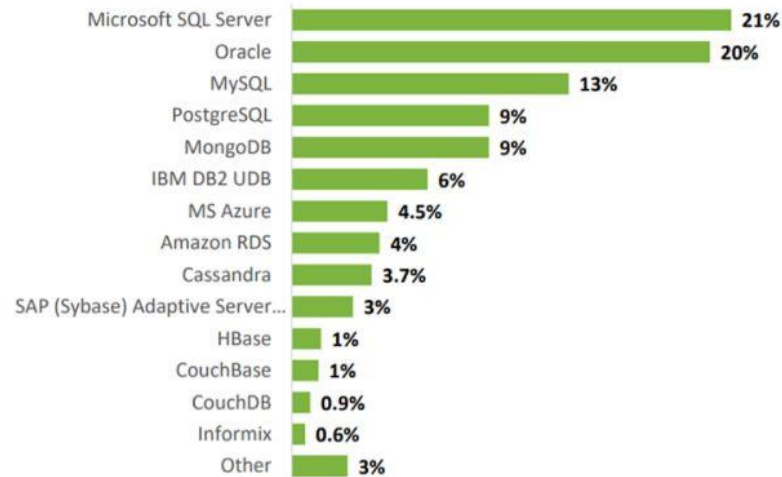
3.3.2 Super Simple Tcp

A Super Simple Tcp [8] egy szerver és kliens közötti, Tcp-Socket alapú kommunikációt megvalósító könyvtár. Ahogy a neve is utal rá, igen egyszerűen használható interfészt biztosít mindkét oldalon, bájtok küldésére és fogadására, kapcsolatok kezelésére. Egyetlen hibája, hogy nem garantálja, hogy egy eseményként érkezik meg a küldött adat vagy egy eseményként nem érkezik több egymás után küldött adat. A kívánt minőségű kommunikáció eléréséhez emiatt némi utómunkára volt szükség.

3.4 Adatbázis

A szerveralkalmazás egy Microsoft SQL Server-hez [12] csatlakozik a Linq könyvtár interfészein keresztül. A Microsoft SQL Server egy relációs adatbáziskezelő rendszer (RDBMS), amely támogatja a tranzakciófeldolgozási, üzleti intelligencia- és

elemzőalkalmazások széles skáláját a vállalati IT környezetekben. Továbbá a Microsoft SQL máig az egyik legelterjedtebb adatbázisfajta ipari környezetben ahogy azt a **3.6. ábra** mutatja.



3.6. ábra - Milyen adatbázisokat használnak az iparban, forrás: [12]

Mivel az alkalmazás Microsoft környezetben készült, a támogatottsága miatt esett rá a választásom, valamint az adatbázis növekedésével jól kihasználhatóak lesznek az MSSQL által nyújtott analízis szolgáltatások és lekérdezés optimalizálási technikák.

3.4.1 System.Linq

A LINQ to SQL [11] a .NET-keretrendszer 3.5-ös verziójának összetevője, amely futási idejű infrastruktúrát biztosít a relációs adatok objektumként való kezelésére.

A relációs adatok kétdimenziós táblázatok gyűjteményeként jelennek meg, ahol a közös oszlopok táblázatokat kapcsolnak egymáshoz.

A Linq programban a relációs adatbázis adatmodellje a C#-ban, de akár más programozási nyelvben kifejezett objektummodellhez van leképezve. Az alkalmazás futtatásakor a Linq lefordítja SQL-re az objektummodell nyelvbe integrált lekérdezéseit, és elküldi azokat az adatbázisba végrehajtás céljából. Amikor az adatbázis visszaadja az eredményeket, a Linq lefordítja azokat, jelen esetben C# objektumokká.

4 Entitások kinyerése nyers szövegből

Ebben a fejezetben részleteiben bemutatom azt az eljárást, amellyel a program tetszőleges adásvételi szerződésből – amelyet nyers, magyar nyelvű szöveggént kap meg – előállítja az adatbázisba menthető rekordokat. Az eljárás részben támaszkodik Deep Learning alapú megoldásokra, ezeknek elméleti hátterét és felhasználásuk menetét egy adott szintű részletességgel levezettem a [Named Entity Recognition \(NER\)](#) fejezetben.

Továbbá bemutatom az összes adat begyűjtését és a felhasználásra való előkészítésük módját, a vizuális analízis eredményével egyetemben, amit a Power BI Desktop alkalmazás segítségével értékeltem ki.

Azon kívül, hogy az adásvételi szerződések ténylegesen természetes nyelven íródnak, nem szabad figyelmen kívül hagyni, hogy egy bizonyos mértékig szerkezetük kötött és számos sablont alapul véve állítható, hogy megjelennek többször ismételt kifejezések, szövegrészek. Például egy személy neve előtt nem valószínű, hogy az fog állni, hogy „adóazonosító”, hanem sokkal inkább olyan kifejezés, mint „vezetéknev”, „családi név” stb. Ezen kifejezések száma még magyar nyelven is véges és a feldolgozás során ezt a tulajdonságot használok ki a jól ismert reguláris kifejezések használatával.

Emellett az attribútumok pontosabb felismerése és kategorizálásának céljából a hagyományos algoritmizálás mellett a program épít a mesterséges intelligencia eszköztárára. Mindezt a NER alkalmazásával, a természetes nyelvfeldolgozás modern úttörőjére, a [BERT](#)-re támaszkodva.

4.1 Modellben felhasznált adatok

A következőkben tárgyalt adatok a NER modell betanítására lettek alkalmazva. A céloom itt az volt, hogy egy minél terjedelmesebb és változatosabb adathalmazom legyen, hogy minél több szóra, minél pontosabb becslést tudjon adni a modell.

4.1.1 Adatok begyűjtése

Szerencsére számos kutatás foglalkozik NER adathalmazokkal, amelyekből néhány magyar nyelven is elérhető, köztük a Szegedi Tudományegyetem Informatikai Intézetének „named entity corpus” -a [15], amely magyar viszonylatban talán a legismertebb, több mint 520 000 szót tartalmaz, minden szóhoz nyelvész szakértők által

rendelt klasszifikáció tartozik. Én mégis egy másik adathalmazt választottam, ez pedig a Hugging Face-en elérhető wikiAnn adathalmaz [16] 'hu' címkéjű részhalmaz. A döntést az indokolta, hogy bár csak fele annyi szót tartalmaz, mint a Szegedi Tudományegyetem adathalmaz, vizuális analízis közben kiderült, hogy teljesebb klassz arzenállal rendelkezik és az egyes klasszokba tartozó szavak számához egy egyenletesebb eloszlás tartozik a wikiAnn esetében, míg a másokban van olyan klassz, ami a szavak 93.18%-át lefedi. Továbbá a wikiAnn egy előre szétválogatott adathalmazt biztosít, azaz külön fájlban kapjuk a tanításra, tesztelésre és validációra szánt részhalmazokat, ami többletmunkától szabadít meg minket.

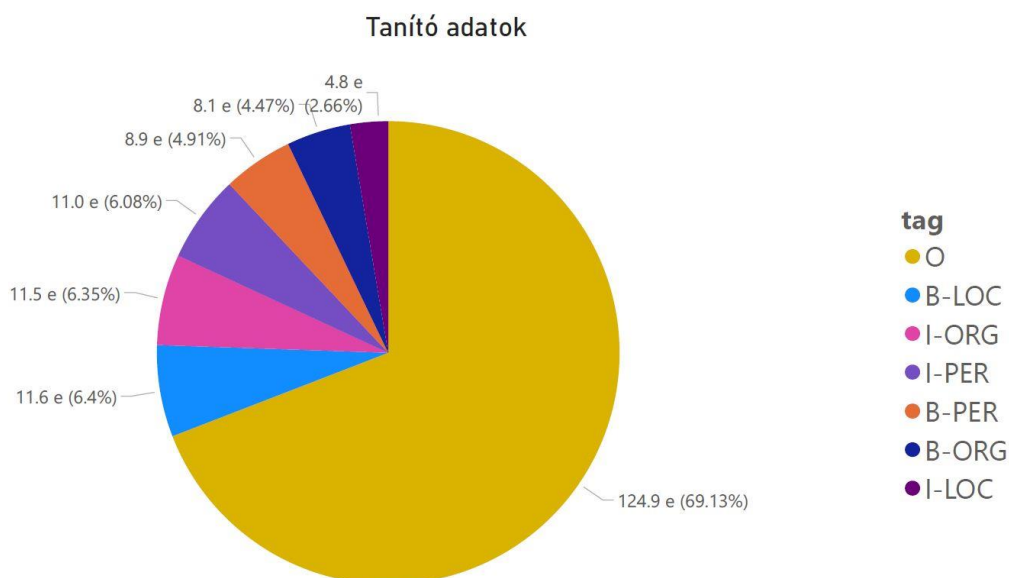
4.1.2 Vizuális analízis

Az adathalmazt CSV formátumban sikerült letöltenem. Először nézzük meg, miket is tartalmaz pontosan. Kezdetben egy sorban három érték található:

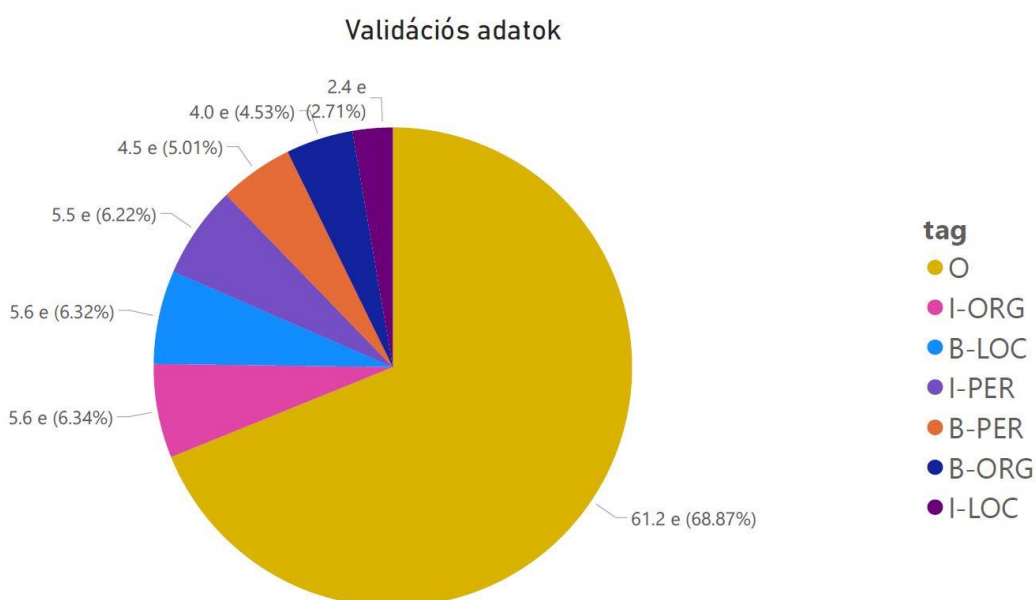
- A sorhoz tartozó azonosítót, amire később már nem lesz szükség.
- Egy tokent, ami egy elnevezés vagy magyar nyelvű szó.
- Egy címkét (tag), ami a klasszt jelöli, ahová a szó be lett osztva:
 - B-PER – Egy személyt jelentő szó vagy kifejezés első szava vagy szótagja
 - I-PER – Egy személyt jelentő szó vagy kifejezés további szótagjai
 - B-LOC – Egy helyszínt jelentő szó vagy kifejezés első szava vagy szótagja
 - I-LOC – Egy helyszínt jelentő szó vagy kifejezés további szótagjai
 - B-ORG – Egy szervezetet jelentő szó vagy kifejezés első szava vagy szótagja
 - I-ORG – Egy szervezetet jelentő szó vagy kifejezés további szótagjai
 - O– Nem tudjuk besorolni

Ezen kívül az adathalmazban lévő szavak sorfolytonosan értelmes mondatokat alkotnak, amiket '.', '?' vagy '!' zár le.

A **4.1. ábrán** és a **4.2. ábrán** látjuk a tanító és validációs adathalmaz eloszlását a klasszok között. Látszik, hogy még így is az 'O' klassz dominál 69,13%-kal, de ez elkerülhetetlen, mivel a nyelv legtöbb szava nem kategorizálható a maradék 6 kategóriába. Hasonló arányok figyelhetők meg a validációs adathalmazon is.

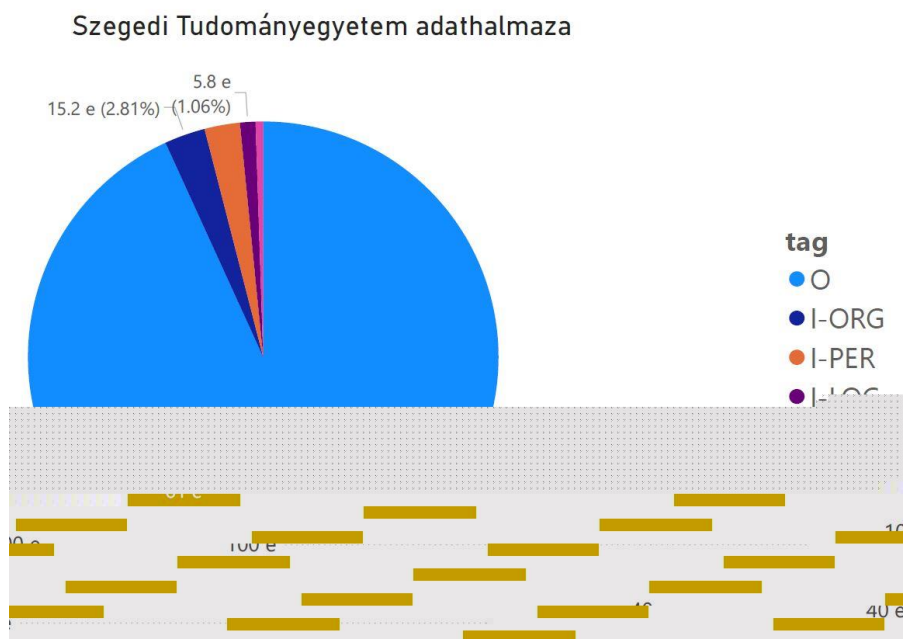


4.1. ábra – tanító adathalmaz eloszlása, Power BI-vel készítve



4.2. ábra – validációs adathalmaz eloszlása, Power BI-vel készítve

Érdekességgént bemutatom a Szegedi Tudományegyetem Informatikai Intézetének adathalmazát a **4.3. ábrán**. Egyből feltűnő, hogy kevesebb klasszt tartalmaz, nem annyira részletes. De ezen kívül az 'O' kategóriás szavak elnyomó többségben vannak, mindössze 7.82% a nem 'O' kategóriás tag és ezek sem kimondottan egyenletesen fordulnak elő.



4.3. ábra – A Szegedi Tudományegyetem adathalmazának eloszlása, Power BI-vel készítve

Végül a **4.4. ábra** egy összehasonlítást mutat a tanító és validációs adatok között. Összesen 269 440 mintánk van, amiből 180 650-et (67%) tanításra használunk fel és 88 790-át (33%) validálásra.

4.4. ábra – Összehasonlítás a tanító és validációs adatok között, Power BI-vel készítve

4.1.3 Adatok előkészítése

Első lépésben a [Pandas](#) könyvtár segítségével betöltöttem az adathalmazt a tanító programba, ami elvégezte helyettem az oszlopok szétválasztását és felcímkézését. Ez után töröltem az esetleges üres sorokat. Kidobtam az azonosítókat és csak a token és tag mezők sorait tartottam meg. Az összes token elejéről és végéről eltávolítottam az esetleges extra szóközöket, végül a tokenek listáját felszeltem mondatokra úgy, hogy minden mondat külön listában legyen, tehát amit kaptunk az egy lista, a szavak listájáról. Ugyanezt tettem a megfelelő tag mezőkkel. Ezzel egy olyan formára hoztam az adatokat, amely a modell tanítását elősegítő könyvtárak interfészeivel már kompatibilis.

4.2 NLP algoritmus

A fejezet ezen részének célja, hogy bemutassa, milyen módon értelmezi az alkalmazás az adásvételi szerződések nyers szövegét. Először szó lesz az alkalmazott Deep Learning alapú NLP technológiákról, a Transformers architektúráról és a BERT modellről, majd a NER modellbecslés használatának és az eredmény feldolgozásának módszerét mutatom be.

4.2.1 A modell finomhangolása

Ahhoz, hogy egy kész csővezetékot kapjak, ami képes természetes nyelvű magyar szöveg kifejezéseinek a klasszifikálására, egy előre tanított BERT modellt kellett finomhangolnom NER specifikus feladatra. A finomhangolás előtt ki kellett választanom a megfelelő előre tanított modellt. Kézenfekvő lett volna a [BERT](#)-ről szóló fejezet végén említett, [huggingface.co](#)-n is elérhető „bert-base-multilingual-cased” modellt használnom, de ez pont a sok nyelvre való felkészítettsége, miatt túl robusztusnak bizonyult, nem tudtam vele használható modellt építeni. Alternatívaként a [huggingface.co](#)-n szintén elérhető „SZTAKI-HLT/hubert-base-cc” [21][22] névre hallgató kizárólag magyar nyelvre felkészített modellt használtam fel.

Kezdetben a modell tanítására Google Colaboratory Notebook-ot használtam és a Google által ajánlott távoli Tesla K80 GPU-t vettem igénybe PyTorch segítségével, ami maximum 12GB RAM-ot biztosít. Mivel saját GPU-val is ennyi memóriát tudok kihasználni, átálltam arra és a lokális fejlesztő és futtató környezetre, mert mindezekon kívül is megbízhatóbbnak és gyorsabbnak ítéltam a Google Colaboratory Notebook esetenkénti lefagyásai miatt.

Ezekkel az erőforrásokkal és az összes tanító és validációs adat felhasználásával a tanítás sajnos beláthatatlan ideig tartott volna, ezért mindkét adathalmaznak csak az egyharmadát használtam fel ténylegesen, ami 60215 tanító és 29596 validációs, bejegyzést jelent.

A betanítás és kiértékelés eredményei az **4.5. és az 4.6. ábrán** láthatóak az egyes tanítási iterációkra (epoch) lebontva.

A tanítás és validálás teljes összesített ideje még kevesebb adattal is 4 óra 47 perc 59 másodperc lett úgy, hogy 3 iterációt ismételtam és a tanulási ráta 5×10^{-5} volt.

Epoch	Training Loss	Valid. Loss	Training Time	Valid. Time
1	0.283284	0.154337	2:57:11	0:06:15
2	0.106069	0.136002	0:44:04	0:07:10
3	0.0536358	0.128757	0:46:38	0:06:41

4.5. ábra – A modell tanulásának adatai az epoch-ok alatt

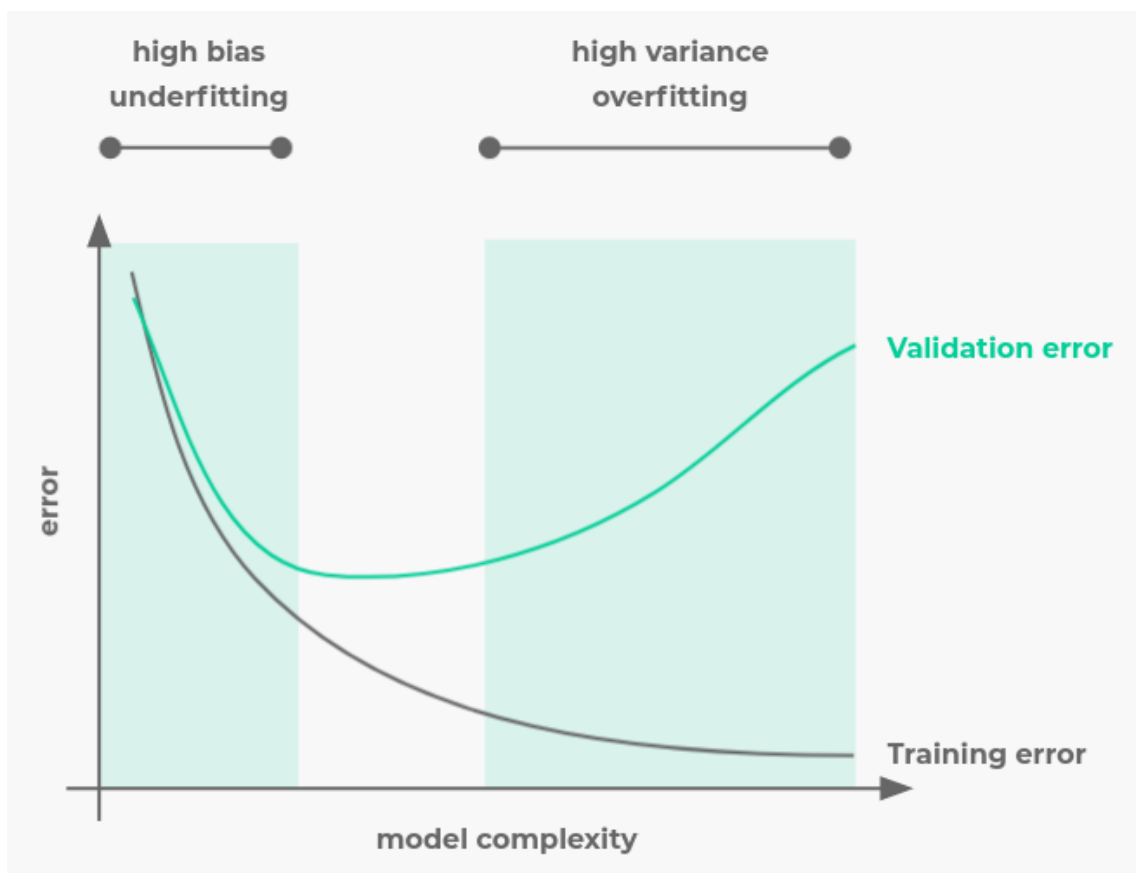


4.6. ábra – Tanulási és validációs veszteségek az epoch-ok alatt, matplotlib.pyplot segítségével kimutatva

Minél kisebb a tanítási és a validációs veszteség (Loss) egyes iterációkban, annál jobb a modell, kivéve, ha a modell túlzottan illeszkedik a betanítási adatokhoz. Ez az over-fitting jelenség. Ilyenkor a modell megjegyzi a tanító adathalmazt és kevésbé képes általánosításra, ennek következtében a validációs veszteség növekszik. A veszteség nem

százalékos, hanem az egyes példákban elkövetett hibák összegzése, vektorok közötti eltérés.

Az **4.6. ábrán** látszik az is, hogy a két veszteség folyamatosan csökken, ahogy a ciklusok ismétlődnek. Meg lehetne nézni még egy negyedik iterációt is, annak reményében, hogy a validációs veszteség tovább csökken, de inkább valószínű, hogy azzal már túlilleszteném a modellt, mivel a validációs veszteség görbe meredeksége a harmadik ciklus végére csökken. Ez a jelenség látható az **4.7. ábrán** is, amivel azt akarom demonstrálni, hogy ott, a középső szakaszban érdemes megállni, ahol a validációs veszteség görbe kezdi elérni a minimum pontját.



4.7. ábra – Over-fitting jelenség demonstrálása, forrás: [23]

4.2.2 Eredmény kinyerése

A modell betanítása után a modellen végzett predikciókat egy Python szkript hajtja végre. Ezt a szkriptet a .NET alkalmazásból futtatom, megadva a bemeneti argumentumait. A szkriptnek 4 bemeneti argumentuma van:

1. Egy kulcsszó, ami megmondja, hogy milyen műveletet akarunk végrehajtani. Jelen esetben ez a 'predict' kulcsszó lesz.
2. A modell elérési útvonala.
3. A speciális karakterektől megtisztított nyers szöveg.
4. A kimeneti fájl, ahonnan az eredményt olvassuk majd.

Az eredményt a szkript JSON formátumban állítja elő a NER csővezeték és a negyedik argumentumban megadott fájlba írja, ahonnan az alkalmazás kiolvassa és objektummá alakítja azt. Ez egy lista, ami tartalmazza a beadott szöveg szavait és a hozzájuk rendelt címkéket és indexeket, már ha a hozzárendelés lehetséges volt.

Konkrétabban egy lista elem a következő információkat hordozza:

- Entity: A címke, amit a tokenhez rendelt a modell („I-PER”, „B-ORG”, stb.).
- Score: egy 0-1 közötti szám, ami azt jelzi milyen mértékben biztos a modell az eredményben.
- Index: Azt mondja meg hányadik szó ez a szövegben.
- Word: Maga a szó vagy token.
- Start: A szó első karakterének pozíciója a szövegben.
- End: A szó utolsó karakterének pozíciója a szövegben.

A listán, a fájlból való betöltés után egy kis utómunkát végeztem, hogy könnyebben felhasználható legyen:

- Minden elem „Index” mezőjét eltávolítottam, mert a szöveg béli pozíciókat a feldolgozás során karakter szinten kezelem.
- Összevontam azokat az elemeket, amiknek a címkéjének második része, vagyis az „I-”, „B-” után lévő része egyezik és indexeik alapján rögtön egymás után következnek a szövegben. Ehhez új kezdő és végpozíciót, valamint egy átlagolt súlyozást állítottam be és csak a címke második részét hagytam meg.
- A súlyokat három tartományba osztottam:
 - $Súly < 0.4$: *Uncertain*, vagyis nem hiszem el az eredményt.
 - $0.4 \leq Súly < 0.8$: *Medium*, vagyis elhiszem, de fenntartásokkal kezelem.
 - $Súly \geq 0.8$: *Certain*, vagyis elhiszem az eredményt.

4.3 Szerződés specifikus adatok

Ezek azok az adatok, amik leírják egy-egy attribútum elvárt formátumát és a kontextust, ahogy egy szerződésben szerepelhetnek, mindezt Regex kifejezések formájában. Ezeket a program futásidőben használja fel a [Regex alapú algoritmus](#) segítségével. A NER modell betanítására nem lettek felhasználva.

4.3.1 Adatok begyűjtése

A célom az volt, hogy minél többfajta adásvételi szerződéshez hozzájussak és az azokban található lényeges attribútumokat, kontextusokkal együtt kilistázzam. Ez igen nehéz feladatnak bizonyult, mivel ilyen speciális célra nem találtam online letölthető adathalmazt, főleg nem magyar nyelven. Ezért arra kényszerültem, hogy az adatgyűjtést kézzel végezzem el.

Először összegyűjtöttem nagyjából 30 féle szerződésmintát főleg a Hvg-Orac Jogkódex [14] adatbázisából, külsős ügyvéd ismerőstől és az interneten publikusan elérhető mintákból. Majd ezeket elemezve listáztam ki minden fontosabb attribútumot, azok elvárt formátumát és a lehetséges kontextusokat, amikben szerepeltek.

4.3.2 Adatok formátuma

A szerződésekből kigyűjtött információ alapján minden attribútumhoz létrehoztam egy később kódból betölthető XML állományt, amelyben felsorolom a következőket:

- *Patterns* - az elfogadott formátumok listája.
- *ContextsAfter* - a kifejezés előtt álló lehetséges kontextusok listája.
- *ContextsBefore* - a kifejezés mögött álló lehetséges kontextusok listája.
- *InvalidsAfter* - a kifejezés előtt álló tiltott kontextusok listája.
- *InvalidsBefore* - a kifejezés mögött álló tiltott kontextusok listája.

Valamint néhány kiegészítő információt, amit feldolgozás közben hasznosítok:

- *Tag* – Az attribútumot azonosító kulcs. Ezek alapján lesznek a programban csoportosítva és ez alapján lehet majd hivatkozni rájuk.
- *PatternCaseInsensitive* – a keresés közben a formátum esetén nincs különbség kis- és nagybetűk között (a kontextus esetében sosincs különbség).
- *OnlyPattern* – csak a formátum számít, a kontextus nem.

Egy ilyen XML fájl formátuma az **4.8. ábrán** figyelhető meg, ahol a családi név attribútum formátuma és kontextusai vannak felsorolva. Az ábrán még észrevehető a *ContextsAfter* listában egy `@{UTONEV}` szöveg. Ez nem egy reguláris kifejezés része, hanem egy saját nyelvtani elem a kontextusok leírásához. Ezt a funkciót és a fájlok felhasználását a [Regex alapú algoritmus](#) fejezetben részletezem, egyelőre csak annyit, hogy a példa kifejezés azt jelenti, hogy a keresett attribútum után egy utónév szerepelhet.

```
<?xml version="1.0" encoding="utf-8"?>
<EntityPattern Tag="CSALADINEV" OnlyPattern="false"
  PatterCaseInsensitive="false">
  <Patterns>
    <string>[A-ZÁÉÖÜŰŐÍ][a-zíéáűőőüó]+</string>
  </Patterns>

  <ContextsBefore>
    <string>családi és utónév</string>
    <string>vezetéknév és keresztnév</string>
    <string>vezetékeve és keresztnéve</string>
    <string>családi és utóneve</string>
    <string>családneve</string>
    <string>családi neve</string>
    <string>vezetéknév</string>
    <string>vezetékeve</string>
  </ContextsBefore>

  <ContextsAfter>
    <string>@{UTONEV}</string>
  </ContextsAfter>

  <InvalidsBefore>
    <string>anyja( születési)? családi és utóneve</string>
  </InvalidsBefore>

  <InvalidsAfter>
  </InvalidsAfter>
</EntityPattern>
```

4.8. ábra - Egy példa a családi névhez tartozó XML állományra

4.4 Regex alapú algoritmus

A Regex alapú algoritmus nyers szöveg béli entitások felismerésére a [Szerződés specifikus adatok](#) fejezetben tárgyalt adathalmazból elő állított XML állományokat

használja fel. Erre a célra egy *RegexEntityMatcher* nevű komponens szolgál, ami létrejöttkor beolvassa ahhoz az attribútumhoz tartozó XML állományt, amilyen típusú entitás a feldolgozás során éppen keresett. Az állományból kinyert formátum és kontextus listát felhasználva, a kiegészítő információkat figyelembe véve, az elemezni kívánt szövegrészletből Regex motor segítségével kikeresi azokat a kifejezéseket, amik megfelelnek az XML állomány által leírtaknak.

Ebbe az eljárásba épül bele később a NER modell csövezeték eredményének felhasználása a súlyok manipulálásával.

Az algoritmus egyik jó tulajdonsága, hogy eredményessége folyamatosan javítható, ha bővítjük a formátumok, kontextusok listáját. Emellett sajnos rontható is, tehát jól át kell gondolni mit veszünk fel a listába.

4.4.1 Attribútumok keresése formátum alapján

A *RegexEntityMatcher* először a szövegben, a formátumokra illeszkedő kifejezésekből állít elő egy listát úgynevezett *MatchResult* objektumokból:

```
public class MatchResult
{
    public string Value { get; }
    public int BeginIndex { get; }
    public int EndIndex { get; }
    public int Weight { get; set; } = 0;
    public string Tag { get; }
    public List<Context> foundContexts = new List<Context>();

    public MatchResult(string value, int beginIndex, int endIndex, string
tag, int weight = 0)
    {
        . . .
    }
    . . .
}
```

Ezekbe az objektumokba elmenti a találat értékét, kezdő és végső indexét (*BeginIndex*, *EndIndex*) a szövegben, valamint az XML fájlból kiolvasott attribútum kulcsot (*Tag*). Minden találat súlya (*Weight*) kezdetben 0, kivéve az *OnlyPattern* jelzőjű attribútumoknak, mert azok alaphoz 1-es súlyt kapnak.

A formátum illesztés közben jön képbe a *PatternCaseInsensitive* jelző. Ha be van állítva, akkor az adott attribútum formátuma a kis- és nagybetűkre nem érzékeny. Ez egy erős eszköz, mert például személy neveket tudjuk, hogy nagybetűvel kezdődnek, ez ebben az esetben elvárás, de mi a helyzet egy bankszámlaszámmal az „IBAN” jelzővel. Ha

szerepel ilyen a szerződésben egyáltalán nem lehetünk biztosak abban, hogy nagy betűvel vagy kis betűvel van feltüntetve. Ezért a bankszámlaszám attribútumot leíró XML fájlban ez a jelző be kell, hogy legyen állítva.

4.4.2 Validálás egyszerű kontextussal

Ha már van egy találat listánk csak formátum alapján, akkor azokat ellenőrizni kell, hogy valóban a betöltött kontextusokba beleillenek-e. Minden megtalált valid kontextus után a találatok súlya 1-gyel növekszik, de ha invalid kontextusban találjuk őket akkor kikerülnek a találatok listájából, elvetjük őket.

Az *OnlyPattern* jelzőjű attribútumok esetén ezekre a validációkra nincs szükség, mindig megtartjuk őket 1-es súllyal.

Azt nevezem egyszerű kontextusnak, amikor a kifejezés előtt vagy után szereplő további kifejezéseket egy sima reguláris kifejezéssel adom meg például:

```
<string>családi( és utó)?név</string>
```

Ennek kiértékelése tényleg egyszerű. Annyit kell tennem, hogy konkatenálom a kifejezést a kontextussal a megfelelő irányból adott elválasztó karakterrel, ami legtöbbször szóköz és erre Regex segítségével rákeresek a szövegben. Ha van találat a megfelelő indexen, akkor a kontextus tényleg ott van a szó közvetlen környezetében.

4.4.3 Validálás összetett kontextussal

Sok esetben egy attribútum kontextusa lehet egy másik attribútum például a vezetéknév – keresztnév esete. Azt gondoltam ezt ki lehetne használni, hogy egyszerűbben és hatékonyabban lehessen szabályokat írni a kifejezések környezetére. Ezért létrehoztam egy apró nyelvtant, hogy kontextusként fel lehessen venni egy másik attribútumot, azonosítója alapján. Ennek szintaktikája:

```
...@{KULCS}...
```

Itt a KULCS helyére egy attribútum azonosító kerül, konvenció szerint nagy betűkkel és a ... helyére pedig bármilyen további Regex kifejezés. Példa:

```
<string>@{VAROS} @{UTCA}</string>  
<string>családi( és utó)?név @{CSALADINEV}</string>
```

Azért a @{ } karaktereket használom, mert ezek nem részeik a Regex speciális karaktereinek így biztonságosan lehet velük dolgozni.

Azokat a kontextusokat, amik szigorúan csak egy $@\{KULCS\}$ kifejezésből állnak speciális kontextusoknak nevezem, de a kiértékelésnél összetett kontextusként kezeltem.

Ennek a nyelvtannak az érvényesítése koránt sem volt egyszerű. Több probléma, is felmerült, például, hogy honnan fogom tudni, hogy a kifejezés környezetében van-e megadott attribútum vagy nincs, valamint egymásra hivatkozó attribútumok esetén végtelen rekurzióba kerülhet a program.

A végső megoldás mindkét problémát kezeli. A lényeg, hogy először csak az egyszerű kontextusok alapján történik validálás, ekkor sok találat súlya megnövekszik. A következő lépésben jön az összetett kontextusok általi validálás, ahol csak az olyan speciális kontextusokat fogadok el, amiknek a súlya nagyobb, mint 0. Ebben a lépésben is több találat súlya megnövekszik. A második lépést addig ismétlem, ameddig van változás a találatok listájában, a súlyok változnak vagy találatok kikerülnek a listából.

Összetett kontextusoknál az illeszkedés eldöntése sem triviális főleg, hogy tetszőleges számú speciális és sima kontextus keveréke lehet. A kiértékelést végül úgy oldom meg, hogy szétdarabolom az összetett kontextust az alapján, hogy egyszerű vagy speciális. Irány függvényében, sorban kiértékelek az alap kifejezésre egyet a listából. Ha illeszkedik, akkor hozzáfűzöm az értékét az eredetihez és úgy veszem, mintha az új lenne az eredeti kifejezés, majd megismétlem a kiértékelést a lista következő elemével.

Egy speciális kontextus kiértékelése már egyszerű feladat. Annyit kell tennem, hogy az azonosítóra illeszkedő találatok listájában megkeresem a nullánál nagyobb súlyú elemeket és index alapján el lehet dönteni, hogy a kontextus a kifejezés közvetlen környezetében van-e.

A jövőben az összetett kontextusok ötletét ki szeretném terjeszteni a formátumok leírására is, mondjuk formátum öröklés formájában.

4.5 A két algoritmus összefonása

Miután rendelkezésre állnak a Regex algoritmus és a NER algoritmus keresésének eredményei is, a célom az volt, hogy a két, formátumában és jelentésében különböző adatszerkezetet együtt érvényesítsem, hogy a program pontosabb találati eredményekkel dolgozhasson tovább.

Ezt úgy tettem, hogy a Regex algoritmus eredményeiből indultam ki és a NER algoritmus eredményei alapján végeztem módosításokat rajtuk. A probléma

bonyolultságát az okozta, hogy nem lehet minden Regex eredményből következtetést levonni, bármilyen NER eredmény alapján, tehát például egy „PER” címkéjű NER eredményt nem lehet összeegyeztetni egy cégnévként felismert szóval, ha az más helyen áll a szövegben. Ugyanakkor, ha ugyan ott állnak, akkor lehet, de azt máshogy kell kezelni, mint az előző esetet.

Az emiatt kezelhetetlenné váló feltételrendszer karbantartására egy szabályrendszert hoztam létre. Szabályokat tudok definiálni úgynevezett *MergeRule* objektumok formájában egy *MergeRuleHolder* objektumban és ezeket a szabályokat sorra végrehajtani minden egyes NER eredményre:

```
class MergeRule
{
    public delegate void Action(List<MatchResult> mrs, NerResult nr);
    public delegate bool Condition(List<MatchResult> mrs, NerResult nr);

    private string[] nerTags;
    private Action action;
    private Condition condition;

    public MergeRule(string[] nerTags, Condition condition, Action action)
    {
        . . .
    }

    public void execute(List<MatchResult> mrs, NerResult nr) {
        if (condition(mrs, nr) && nerTags.Contains(nr.entity))
        {
            action(mrs, nr);
        }
    }
}
```

Egy *MergeRule*-nak először meg kell adni milyen NER címkéjű eredményekre vonatkozik a szabály (*nerTags*), majd egy feltételt, amely megmondja, hogy minek kell teljesülnie a szabály érvényesítéséhez (*Condition*), például a Regex eredmény (*MatchResult*) és a NER eredmény (*NerResult*) meg kell, hogy egyezzen. Végül egy műveletet adok meg, ami végrehajtja a változtatásokat a Regex eredmények listájában (*Action*), ezt a *MergeRule* csak akkor hajtja végre, ha a feltétel igaznak bizonyul.

Úgy gondolom, hogy ez a megoldás jól skálázható, ha sok szabályt szeretnék létrehozni és mindet különböző címkékre specifikálni, más-más feltételekkel és akciókkal. Az összefűzés végén a Regex eredmények listája és azok elemei olyanná módosulnak, hogy figyelembe veszik a NER algoritmus kimenetelét.

5 Az alkalmazások részletes elemzése

Ebben a fejezetben a rendszer megvalósításának részleteit fogom bemutatni. Szó lesz a kliens és szerver alkalmazások architektúrájáról, valamint a több ötletet igénylő funkcionális egységeik kivitelezéséről, mint az adatok megjelenítése, kommunikáció létrehozása, autentikáció és autorizáció, szerződések analizálásának menete magas szinten és még sok más. Ezen kívül bemutatom a szerver által használt adatbázist.

5.1 Kliens alkalmazás

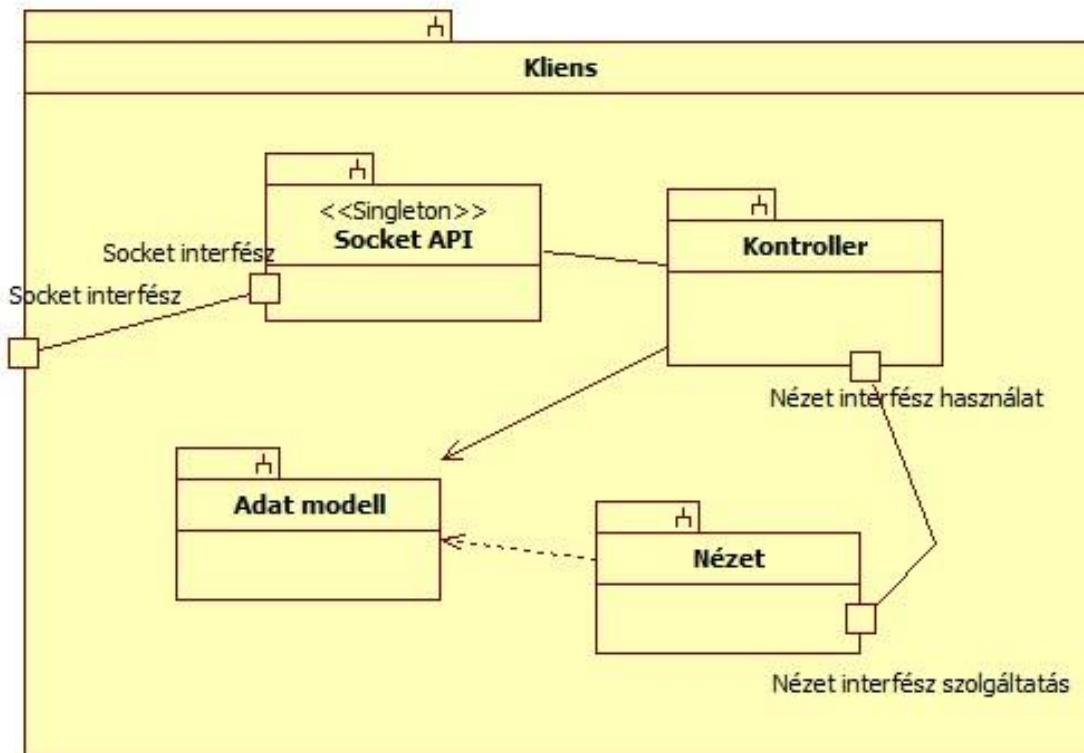
A kliens alkalmazás feladata, hogy egy jól kezelhető interfészt biztosítson a felhasználónak, amelyen keresztül lehetővé teszi az adásvételi szerződések beolvasását és felküldését a szerverre, valamint a már előre beolvasott szerződések adatainak lekérését és megjelenítését. A kliens alkalmazás nem áll kapcsolatban adatbázissal, ehelyett egy-egy adatot konfigurációs fájlból tölt be, mint például a szerver port száma.

5.1.1 Felépítése

Az alkalmazás MVC (Modell View Controller) architektúrát követ, kiegészítve egy Socket alapú kommunikációs API-val. Minden **Nézet** egy ablaknak felel meg, a **Kontroller** felelős az üzleti logika megvalósításáért, az **Adat Modell** pedig a szerverről lekért adatok tárolásáért, valamint a felhasználó autentikációs és autorizációs adatainak számontartásáért.

A nézetek a különböző felhasználói események hatására meghívják a kontroller megfelelő futási egységét, amely vagy kérést intéz a szerverhez a **Socket API**-n keresztül, vagy műveleteket végez az Adat Modellen. A Socket API egy singleton tervezési mintát követő egység, amely folyamatosan hallgatózik egy **Socket interfészen** és ha van a socket portjára érkező üzenet akkor egy eseményt tüzel, majd meghívja a megfelelő üzleti logikát. A kliens magas szintű architektúráját, az előbb elmondottak mellett a **5.1. ábra** mutatja be.

A szerver és a kliens Socket API-ja nagyon hasonló, ezekről bővebben a [Socket API és kommunikáció](#) fejezetben számolok be.



5.1. ábra - A kliens alkalmazás felépítése, MagicDraw segítségével készítve

5.1.2 Nézet interfész

Események a szerveroldalról is bármikor jöhetnek. Egy ilyen esemény bekövetkeztekor a Socket API a kontrollerhez fordul, hogy végrehajtsa a kért műveleteket és értesítse az összes Nézetet, hogy az esemény hatására állapotváltozás történt. Ehhez minden Nézet egy interfészt valósít meg a kontroller számára, ahol a következő interakciók vannak definiálva:

- **Connected()**: azt jelzi a Nézetek számára, hogy a kliens és a szerver között kapcsolat jött létre.
- **Disconnected()**: azt jelzi a Nézet számára, hogy a kliens és szerver közötti kapcsolat felbomlott.
- **HandleMessage(message)**: azt jelzi a Nézet számára, hogy a szerverről üzenet jött. Ennek tartalmát megkapja a Nézet és a tartalom alapján változást idéz elő a felhasználói felületen.
- **IsActive()**: erre a hívásra a Nézet visszaadja, hogy aktív állapotban van-e. Ez azért fontos, mert egy már nem aktív nézetet a kontroller eltávolít a nézetek listájából.

- **ClientLoggedIn()**: azt jelzi a Nézet számára, hogy a felhasználó sikeresen bejelentkezett.

5.1.3 Adatok megjelenítése

Az adatok megjelenítésére egy *DisplayForm* nevű nézet szolgál, ami az adatokat táblázatosan jeleníti meg. Az adattípusok lista nézetben jelennek meg és minden adattípushoz külön táblázat tartozik. Ezen felül, egy táblázat sorára kattintva egy újabb *DisplayForm* jelenik meg, ami csak a kiválasztott sorhoz tartozó információkat tartalmazza.

Amikor kiválasztunk egy adattípust a kliens lekéri a szervertől a hozzá tartozó adatokat, majd elmenti a memóriába, hogy a későbbiekben ne kelljen újra lekérni. Viszont, ha a szerver jelez, hogy az adatbázis tartalma megváltozott, a kliens újra lekéri a kiválasztott adattípushoz tartozó adatokat.


A feladat nehézsége abban rejlett, hogy hogyan mondjuk meg a szervernek, hogy melyik *DisplayForm* ablaknak milyen adatokat kell küldeni. Megoldásként azt találtam ki, hogy minden *DisplayForm* ablakhoz rendeljek egy elérési útvonalat, hasonló, mint a webalkalmazások által használt végpontok. Ezt az útvonalat küldi el a kliens a szervernek, ami visszafejti azt, és végrehajtja a megfelelő lekérdezést, majd az eredményt az útvonallal együtt elküldi a kliensnek. A kliens az útvonalból tudja meg, hogy melyik ablak, melyik adattípusához tartoznak a kapott adatok. A módszert egy példán keresztül demonstrálom.

Szeretnénk lekérni a 73-as azonosítójú személyhez tartozó összes szerződést. Ebben az esetben az útvonal így néz ki: „/szemelyek/73/szerzodesek”. Ehhez két ablakra van szükségünk. Az első ablak útvonala a „szemelyek” menüpontban állva „/szemelyek”, majd a 73-as azonosítójú személyt kiválasztva egy második ablak nyílik, amelynek útvonala „/szemelyek/73”. Itt kiválasztva a „szerzodesek” menüpontot, kapjuk a kívánt útvonalat. A kapott adatokat csak az az ablak fogja megjeleníteni, amelyiknek aktuális útvonala egyezik a szervertől kapott válaszban talált útvonallal.

5.1.4 Képszedet

Ebben a részben képek segítségével mutatnám meg a kliens felhasználói felület elemeit. A **5.2. ábra** a bejelentkezési formot mutatja az alkalmazás indulásakor, míg a **5.3. ábra** az alkalmazás kezdőképernyőjét, sikeres bejelentkezés után, ahol látszik, hogy

előre be van töltve néhány szerződés és a feldolgozás fut a szerveren. A **5.4. ábrán** az adatokat megjelenítő ablakok láthatóak. A főablak a szerződések általános információi menüpontját mutatja, az „ingatlan” kifejezésre szűrve, míg a második a 20998-as azonosítójú szerződés személyeit.



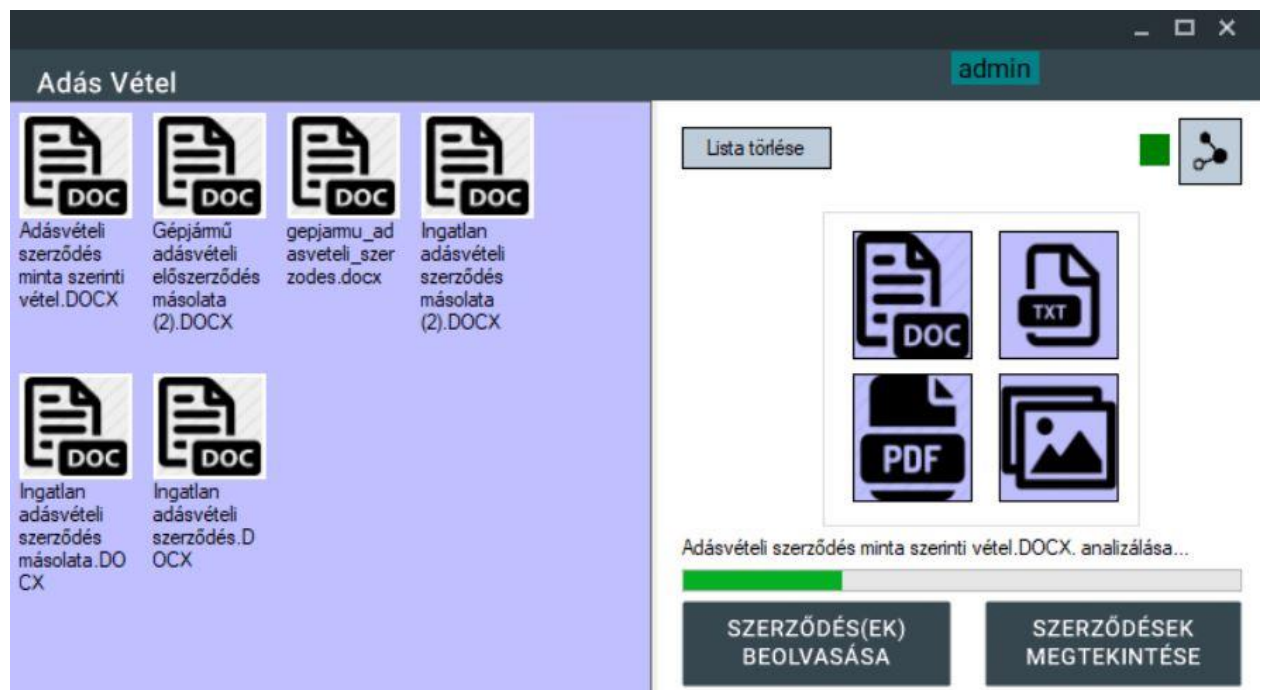
Üdvözljük

Felhasználónév:

Jelszó:

BEJELENTKEZÉS

5.2. ábra - Bejelentkező ablak



Adás Vétel admin

Lista törlése

Adásvételi szerződés minta szerinti vétel.DOCX

Gépjármű adásvételi előszerződés másolata (2).DOCX

gepjármu_ad asveteli_szer zodes.docx

Ingatlan adásvételi szerződés másolata (2).DOCX

Ingatlan adásvételi szerződés másolata.DOCX

Ingatlan adásvételi szerződés másolata.DOCX

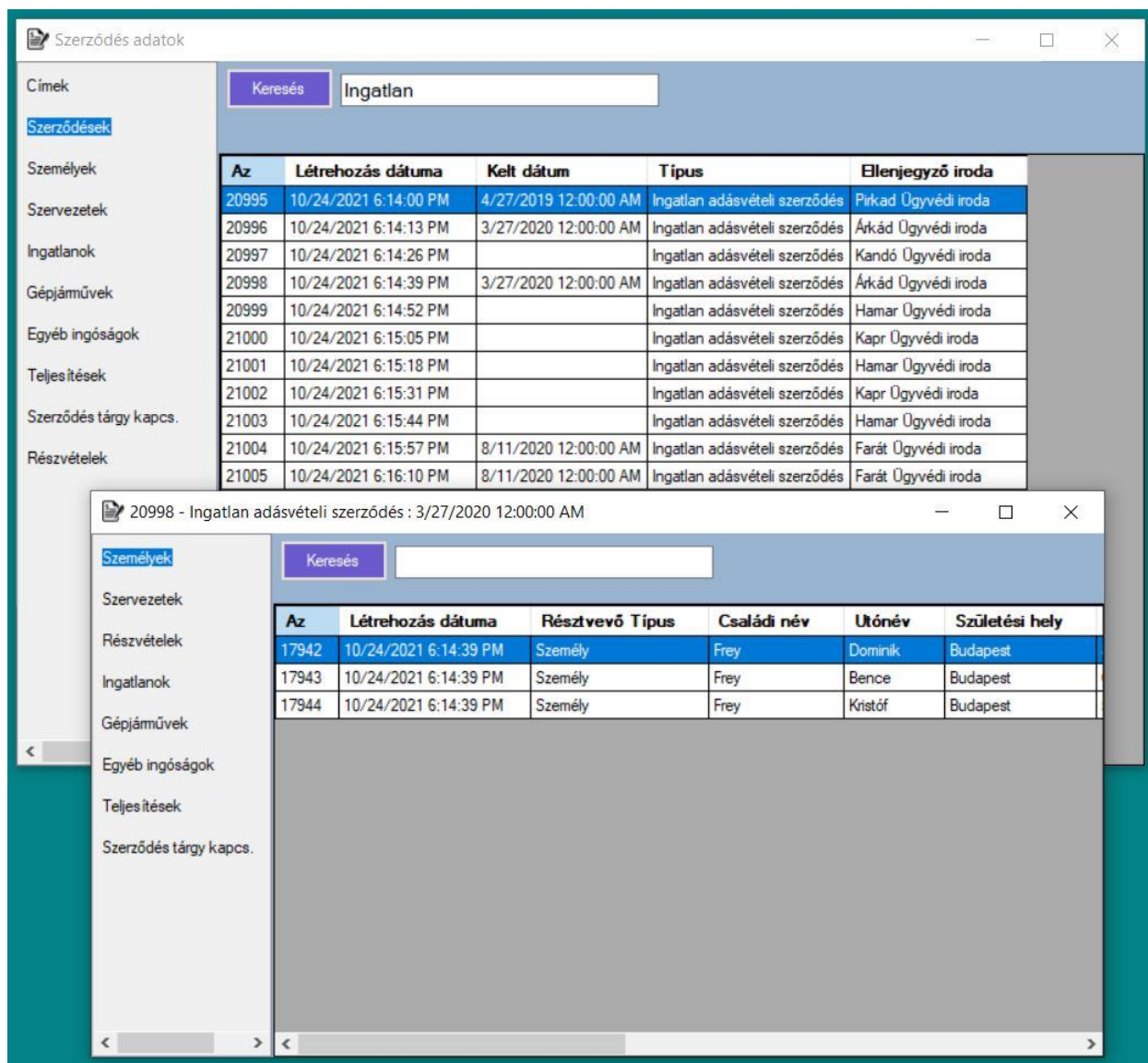
Ingatlan adásvételi szerződés.DOCX

Adásvételi szerződés minta szerinti vétel.DOCX. analízálása...

SZERZŐDÉS(EK) BEOLVASÁSA

SZERZŐDÉSEK MEGTEKINTÉSE

5.3. ábra - Kezdőablak



5.4. ábra - Böngésző ablak

5.2 Szerveralkalmazás

A szerver egy igen sokrétű alkalmazás:

- Folyamatosan menedzseli a kliensekkel való kapcsolatokat a **Socket API**-ján keresztül.
- Authentikációt és autorizációt kezel.
- Analizálja a kapott természetes nyelvű szöveget, majd a kinyert adatokat egy adat modellbe szervezi, amit adatbázisba ment.
- Végül a kliens oldalról kért adatokat lekérdezi majd elküldi.

Ezen kívül a szerver maga is egy ablakos alkalmazás, a fejlesztés megkönnyítése érdekében. A felhasználói felületén egy log megjelenítő található, valamint új felhasználó felvételére és az adatbázis törlésére van lehetőség.

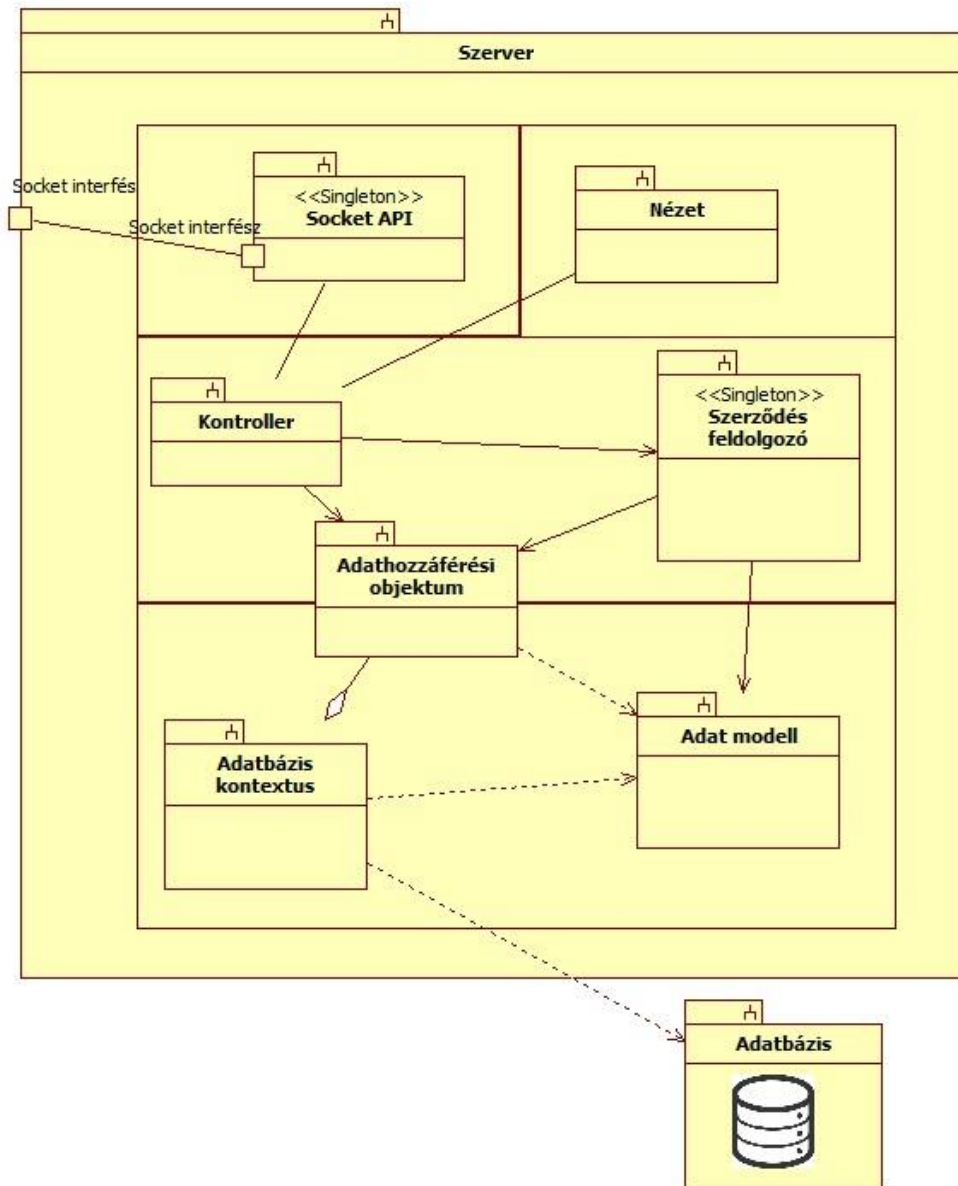
5.2.1 Felépítése

A klienshez hasonlóan a szervert is négy rétegre lehet bontani, valamint az MVC architektúra itt is megfigyelhető, bár a klienssel ellentétben, itt a gyakorlatban csak egy **Nézet** van és a **Kontroller** ehhez nem interfészen keresztül kapcsolódik, hanem közvetlen asszociációja van egyes elemeire. Megjelenik egy **Szerződés Feldolgozó** komponens, ami singleton tervezési mintát követ és minden kapott szerződés esetén, külön szálon végzi el az időigényes feladatokat.

Az adathozzáférési réteget egy **Adathozzáférési Objektum** választja el az üzleti logikától. Az Adathozzáférési Objektum minden adatbázis specifikus hívást elrejt egy jól átlátható, beszédes interfész mögé a [System.Linq](#) könyvtár segítségével.

Úgynevezett „Linq Query Expression”-ök segítségével intézhetünk lekérdezéseket az adatbázishoz az **Adatbázis Kontextuson** keresztül, amely végrehajtja a Linq Query Expression -> SQL konverziót.

Ezen kívül a tényleges objektum modell <—> relációs adatmodell leképezést ez a komponens végzi, az SQL szerverrel való kapcsolat nyitással és bontással egyetemben. A szerver magas szintű architektúráját, az előbb elmondottak mellett a **5.5. ábra** mutatja be.



5.5. ábra - A szerver alkalmazás felépítése, MagicDraw segítségével készítve

5.2.2 Szerződések analízálása

Ha a szerverre beérkezik egy elemzésre szánt szerződés, azt a Kontroller először egy *Task* objektumba csomagolja, ami rögzíti a kérés üzenet tartalmát és a klienst, akinek a feldolgozás közben állapotjelzést kell majd adni. A *Taskot* egy várakozó sorba állítja, amelyből a másik szálon futó Szerződés Feldolgozó egység folyamatosan veszi ki a *Task*-okat és dolgozza fel őket, ha a sor nem üres. Egy *Task* elvégzése előtt és után a Feldolgozó egység üzeneteket küld a kliensnek, hogy jelezze hogyan áll a feldolgozás.

A nyers szöveg a Szerződés Feldolgozó egység futási láncán a következő műveleteken megy keresztül.

Szövegfeldolgozás:

Első lépésben egy *TextAnalyzer* nevű komponens veszi át, ami elvégzi a szöveg megtisztítását a speciális karakterektől (.?!,@,&,{,},\$,ß, stb.). Ez után lefuttatja a NER becslést végző Python szkriptet egy párhuzamos szálból, aminek a kimenetéről JSON formátumban beolvassa a token – címke (tag) összerendelések listáját. Eközben párhuzamos végrehajtással futtatja a Regex algoritmust az egész szövegre, majd a két feldolgozás egy ponton a kódban bevárja egymást és eredményeiket összefűzi [A két algoritmus összefonása](#) fejezetben megismert *MergeRule*-ok alapján:

```
public void analyzeText(string data, string filename)
{
    ...

    //Adatok formázása
    text = formatData(data);

    //NER eredmények kinyerése párhuzamos szálból
    NerExecutor nerExecutor = new NerExecutor(text);
    nerExecutor.start();

    //Regex alapú keresés eredményeinek kinyerése
    matchesDict = getAllMatches(text);

    //Várunk, amíg a NER végrehajtott
    nerExecutor.waitForExecutionEnd();
    nerResults = nerExecutor.Results;

    //Eredmények összefűzése
    mergeNerIntoRegexMatches();

    ...
}
```

Megjegyzendő, hogy a korábbi próbálkozásaimban egy más megközelítést alkalmaztam, amikor is a program nem alkalmazta a Regex algoritmust az egész szövegre a feldolgozás elején, hanem csak a szöveg egyes, éppen lényeges részeire a futás különböző szakaszaiban, így csak akkor állt rendelkezésre eredmény, amikor kellett és csak annyi amennyi kellett, hogy spóroljak a futásidővel. Ezzel a megközelítéssel az volt a baj, hogy megnehezítette a NER algoritmus eredményeivel való összekombinálást és mivel a NER végrehajtása futásidőben amúgy is szűk keresztmetszet, érdemesnek tűnt inkább az egész szöveget rögtön feldolgozni és a következőkben bemutatott

komponensek egyszerűen csak kivehetik az eredmények listájából azokat, amikre szükségük van.

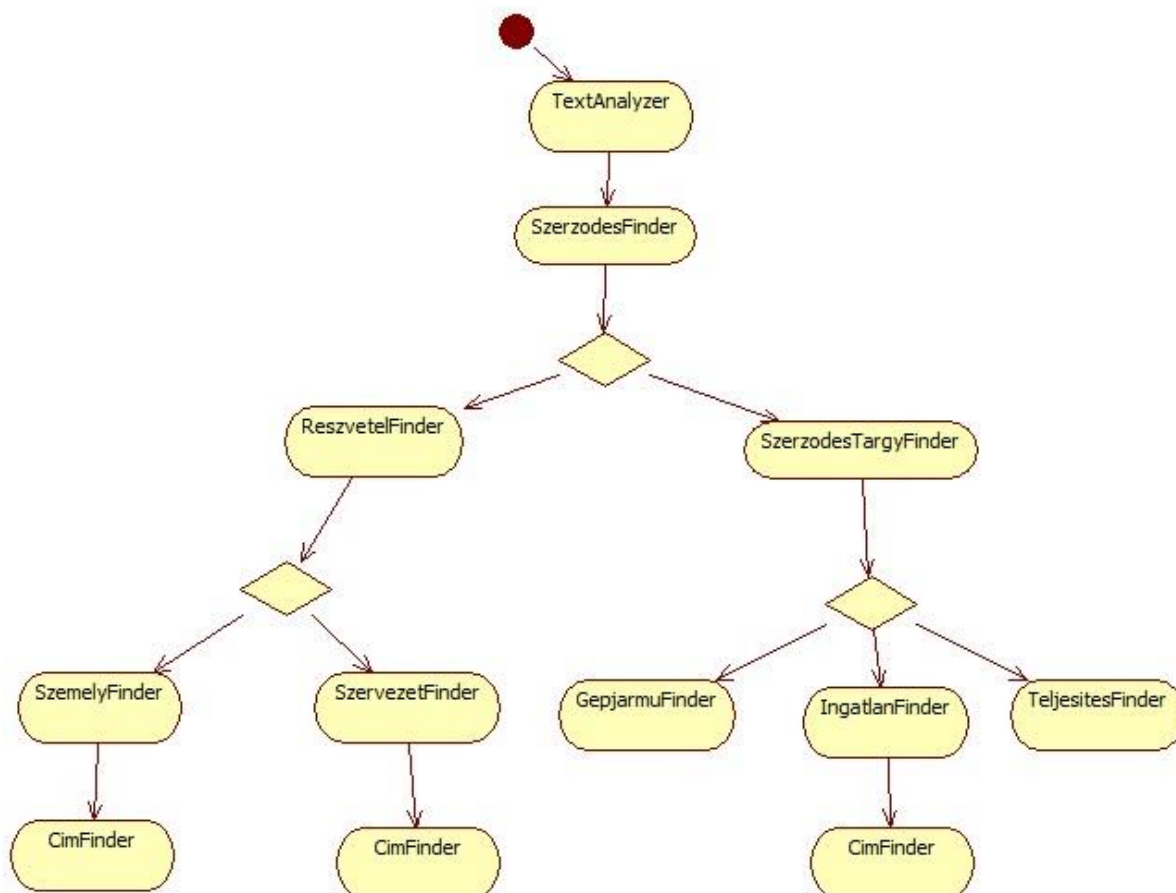
Eredmények csoportosítása, hívási fa:

Az eredmények birtokában a feldolgozó szál bekerül a *Finder* komponensek hívási fájába. A *Finder* komponensek olyan singleton osztályok, amik a szöveg egy adott blokkjában keresik, egy adott adattípus összes előfordulását és azok attribútumait. Mindegyik *Finder* kivág a szövegből egy darabot és átadja a következő *Finder*nek, hogy abban keressen tovább. A következő *Finder* osztályok léteznek:

- *SzerzodesFinder*: a szerződés általános adatait kereső elem.
- *SzemelyFinder*: egy személy adatait kereső elem.
- *SzervezetFinder*: egy szervezet adatait kereső elem.
- *ReszvetelFinder*: egy személyt vagy szervezetet keres egy adott szövegrészben és eldönti, hogy eladó vagy vevő szerepet töltenek be.
- *IngatlanFinder*: egy ingatlan adatait kereső elem.
- *GepjarmuFinder*: egy gépjármű adatait kereső elem.
- *SzerzodesTargyFinder*: egy ingatlant vagy gépjárművet keres egy adott szövegrészben, a teljesítés tárgyával egyetemben.
- *CimFinder*: egy címet kereső elem.
- *TeljesitesFinder*: a szerződés teljesítésének feltételeit kereső elem.

A *Finder* osztályok így egy, a **5.6. ábrán** látható hívási fát képeznek, ahol az osztályok a csomópontok és az átmenetek az egyes hívások paraméterrel ellátva, ahol a paraméter a nyers szöveg egy intervalluma. A gyökér csomópont a *SzerzodesFinder* osztály, ami egy szerződés modell objektumot állít elő és amitől minden további modell objektum közvetlen vagy közvetett függeni fog.

Az elágazások az ábrán azt jelentik, hogy az elágazó részfákat az adott csomópont balról jobbra való sorrendben tetszőleges ismétléssel hívhatja, akár egyszer sem.



5.6. ábra - A Finder komponensek hívási fája, MagicDraw segítségével készítve

Fontos megjegyezni, hogy a hívási fát a program mélységi bejárással járja be és felülről az első elágazás után bármelyik tetszőleges ág lefuthat többször, ha az a helyzet áll fent, hogy az adott adattípusból több fedezhető fel a szerződésben.

Ahogy a program végigfut a hívási fán, a csomópontokban létrejönnek a már adatbázisba menthető objektumok, amiknek attribútumai szövegfelismerő algoritmusok eredményeiből kapnak értéket. Ezek az objektumok egymással a hívási fa alakjának megfelelő függőségben állnak. Minden csomópont legutolsó lépése a találatok validálása, annak vizsgálata, hogy létezik-e már ilyen találat, majd sikeres és egyedi találat esetén az adatbázisba való mentés.

5.2.3 Authentikáció és Authorizáció

A szerver feladatai közé tartozik, hogy a felhasználókat azonosítsa a szerveren tárolt információk védelme érdekében.

Hogy ezt megvalósítsa a felhasználókról három információt tárol az adatbázisban:

- Username: a név, amely a felhasználót azonosítja.
- Password: a titkos jelszó, amely nélkül a felhasználó nem tud belépni.
- Authority: a felhasználó jogköre, ami lehet:
 - HIGH: a felhasználó küldhet szerződéseket a szerverre feldolgozás céljából és le is kérheti a tárolt szerződések adatait.
 - LOW: a felhasználó nem küldhet a szerverre adatot csak lekérdezést hajthat végre.
 - NONE: a felhasználónak semmilyen művelethez sincs jogköre.

A bejelentkezés és az autentikáció a következőképpen zajlik:

A kliens a szerverhez való csatlakozás után küld egy „login” üzenetet, melyben közli a felhasználó nevét és jelszavát. Fontos, hogy a jelszót SHA256 algoritmus által hash-elt formában küldi el a szervernek. Az üzenetet megkapva a szerver azonosítja, hogy a kapott bejelentkezési adatok megfelelőek-e. Ha nem, akkor hibaüzenetet küld vissza, amiben közli a visszautasítás okát.

Ha a bejelentkezési adatok rendben vannak, akkor visszaküldi a felhasználónak a „login” üzenetet kiegészítve annak jogkörével és egy generált úgynevezett 'Session Id'-val, ami a bejelentkezési sessiont azonosítja majd és minden kliens kérésnél ettől a felhasználótól, ezt az azonosítót várja a kérésbe ágyazva. Így dönti el, hogy valóban a jogosult felhasználó küldi a kérést. A válasz előtt az adott porthoz, ahonnan a „login” üzenet jött, hozzárendeli a felhasználó adatait és a „Session Id”-t.

Amikor valamelyik felhasználótól üzenet érkezik, a szerver először ellenőrzi a „Session Id”-t, majd a kéréshez tartozó autorizációs szintet összeveti a felhasználóéval. Csak akkor hajtja végre a műveletet, ha mindkettő megfelel az elvártaknak.

5.2.4 Lekérdezések visszafejtése

Ahogy a kliens alkalmazásnál az [adatok megjelenítése](#) fejezetben láthattuk, a szerver egy klientsől érkező adatlekérés esetén egy útvonalat kap, amely alapján el kell döntenie, milyen adatbázis lekérdezéseket hajtson végre. Ebben a részben ennek a menetét részletezem.

Az üzleti logikában minden használni kívánt végponthoz hozzárendeltem az Adathozzáférési Objektum egy függvényét egy *RouteContainer* nevű komponenssel. Hogy könnyítsem a dolgom Regex kifejezéseket használtam konkrét útvonalak helyett.

A RouteContainer, ha megkap egy útvonalat a Regex motor segítségével kideríti, melyik mintára illeszkedik és lefuttatja a hozzá tartozó függvényt a megfelelő paraméterekkel. Ez a függvény tulajdonképpen egy lekérdezés, tehát a találatok listáját kapjuk meg.

Az eredmény itt még nincs küldhető állapotban, mivel az Adathozzáférési Objektum interfészei objektum modelleket adnak vissza, relációs modellből konstruálva és ugyanilyen objektum modell formában szeretnénk visszakapni a kliens oldalon is. Mivel Tcp-Socket-en keresztül csak szöveget vagy bájt tömböt lehet küldeni, a következőt csináltam:

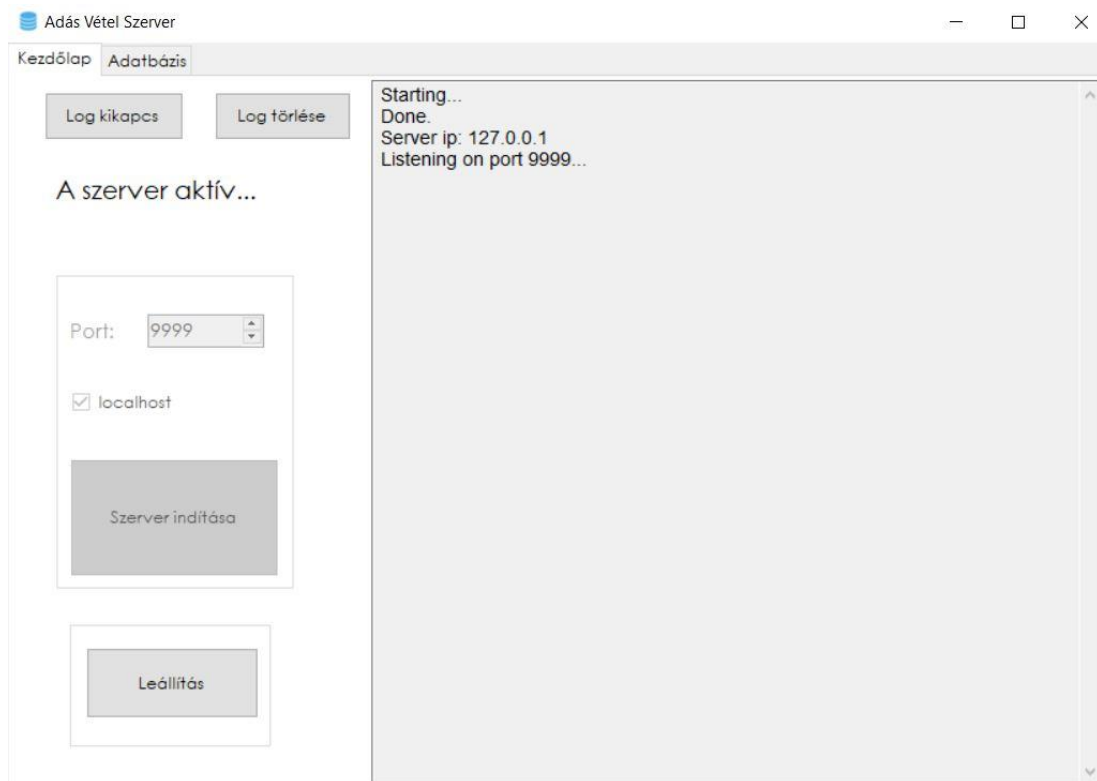
Minden találatot egyesével JSON bájt tömbbé sorosítottam, majd egy listába szerveztem.

A JSON bájt tömböket becsomagoltam egy-egy üzenet objektumba, amik további információkat tartalmaznak az üzenetről, például az üzenet típusát vagy, hogy hányadik találat jön. Végül az üzenet objektumokat szintén JSON bájt tömbbé sorosítottam, majd egyesével elküldtem a kérést intéző kliensnek. A kliens fogadáskor kicsomagolja az üzenetet, majd egy következő lépésben a találatot is visszaalakítja objektum modellé.

Már rájöttem, hogy a jövőben jobb lenne az egyes találatokat egyszerre, egy üzenetben küldeni, hogy csökkentsem az adatforgalmat, ami a kiegészítő, akár redundáns információkkal keletkezik. Bár a megoldás – ezt leszámítva – így is működőképes.

5.2.5 Képszedet

Ebben a részben képek segítségével mutatnám meg a szerver felhasználói felület elemeit. A **5.7. ábra** mutatja a kezdő ablakot, ahol a szervert indíthatjuk vagy állíthatjuk le. Itt olvasható a futás közben generált log. A **5.8. ábra** pedig egy felhasználó regisztrációjának form-ját mutatja, valamint egy gombot az adatbázis törlésére, amire a fejlesztés során volt szükség.



5.7. ábra – A szerver kezdőképernyője

Adatbázis törlése

5.8. ábra – Regisztrációs form

5.3 Socket-API és kommunikáció

Ebben a részben a szerver és kliens közötti kommunikáció megvalósítását mutatom be. Szó lesz a már említett [Tcp-Socket](#) technológia segítségével megvalósított API-ról és az üzenetek létrehozásáról és értelmezéséről mind a szerver, mind a kliens oldalon.

5.3.1 Socket-API

Az Tcp-Socket feletti API-t a [Super Simple Tcp](#) nevű könyvtár segítségével alakítottam ki. A könyvtár úgy működik, hogy SimpleTcpServer és SimpleTcpClient objektumokat kínál fel, amik eseményeire saját metódusokat iratkoztathatunk fel. Ezek az események a következők:

- Szerver esetében:
 - ClientConnected() - egy klienssel létrejött a kapcsolat.
 - ClientDisconnected() - egy klienssel lebomlott a kapcsolat.
 - DataReceived() - egy klientsől adat érkezett.
- Kliens esetében:
 - Connected() - a kliens csatlakozott a szerverhez.
 - Disconnected() - a kliens és szerver közötti kapcsolat lebomlott.
 - DataReceived() - a szervertől adat érkezett.

Ezen kívül mindkét oldalon rendelkezésre állnak Send() függvények, amikkel egyszerű bájtfolyamok küldhetőek a megadott portra.

Viszont a könyvtár önmagában nem kezeli az üzenet „framing”-et, ami azt jelenti, hogy küldéskor-fogadáskor nincs garantálva az üzenethatárok betartása. Tehát előfordulhat, hogy két üzenet egy eseményként érkezik meg vagy egy üzenet kettő vagy több eseményben szétosztva. Ezzel szembenézve döntöttem úgy, hogy megírom a saját API-mat mindkét oldalon a Super Simple Tcp felé építve, ami kezeli az üzenet „framing”-et és küldéskor [előre deffiniált üzenet objektumokat](#) küldhetünk egyszerű bájtfolyam helyett.

Az üzenet „framing” kezelése:

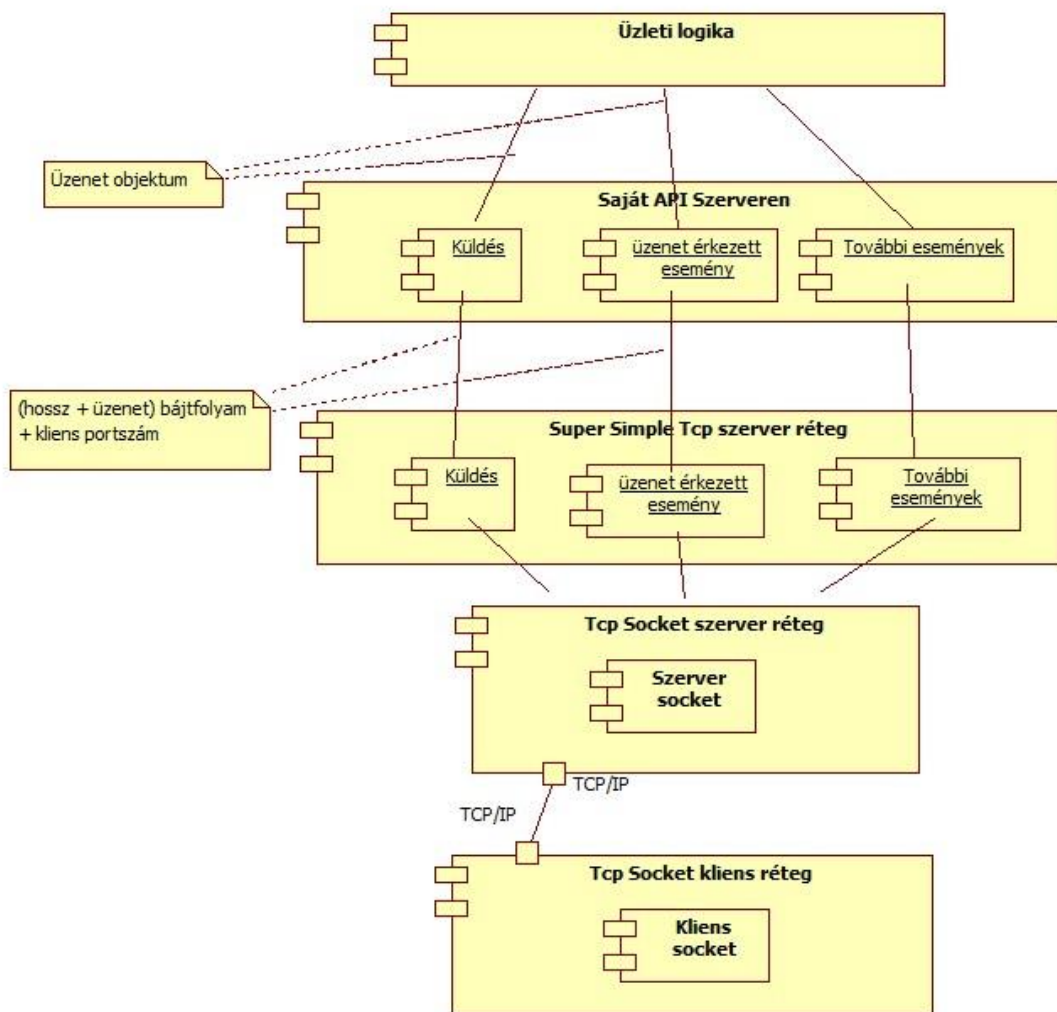
- Az üzenet „framing” kezelése nem egyszerű feladat programozás szempontjából. Az elve az, hogy minden üzenet elejére küldés előtt egy fix 4 bájtnál méretű előjeles integer illeszték, ami megadja a tényleges üzenet hosszát bájtokban. A fogadó

kiolvassa az üzenet első 4 bájtját és innentől tudja, hogy hány bájtnyi tényleges üzenetet kell még kiolvasnia a folyamból. Ha a folyam végére ért és még nincs meg az összes bájt, akkor az állapotot eltárolja és megvárja a következő eseményt, hogy megkapja a maradékot. Ha még nem ért a folyam végére, de megvan az összes szükséges bájt, akkor feldolgozza az üzenetet, majd megismétli az egész műveletet addig, amíg a folyam végére nem ér.

Objektumok küldése és fogadása:

- Küldéskor az előre definiált üzenet objektumokat hasznos adattal feltöltve JSON bájt tömbbé sorosítom majd a bájttömb méretét hozzáillesztem az elejéhez. Ekkor készen áll a küldésre a Super Simple Tcp-n keresztül.
- Fogadáskor az API-n végrehajtódik az üzenet „framing”, majd a feldolgozásra kapott üzenetet kicsomagoljuk a JSON bájt tömbből az üzenet objektumok őssosztályába, amely tartalmazza az üzenet típusát. A típus alapján eldönthető, hogy milyen fajta üzenet objektumba kell kicsomagolni a következő lépésben.

A **5.9. ábra** Az üzenet küldés és fogadás során beérkezett események kezelését mutatja be rétegekbe szedve. A legfelső réteg az üzleti logika, ahol üzenet objektumokat dolgozunk fel. A második rétegben az üzenet objektum <-> bájtfolyam leképezés zajlik, ami magában foglalja az üzenet hossz és a „framing” kezelést. Valamint egy kliens port szám <-> kliens modell + felhasználó leképezés is végbemegy. A kommunikáció további lépéseiről a Super Simple Tcp könyvtár gondoskodik.



5.9. ábra - Socket üzenet küldés és fogadás rétegei, MagicDraw segítségével készítve

5.3.2 Definiált üzenetek

A funkciók megvalósításához elég volt 6 darab különböző jelentéssel és kiegészítő információkkal bíró üzenet objektumot definiálnom, de a számuk bármennyig növelhető. Mind a 6 objektum egy közös ősből származik, aminek *MessageBase* a neve. A *MessageBase* tartalmazza a „Session Id”-t és az üzenet típusának azonosítóját, ami alapján JSON bájt tömbből való kicsomagolás során eldönthető milyen típusú üzenetről van szó. Az üzenetek a következők:

- **GetRecordMessage:** ezzel az üzenettel kér le a kliens adatot a szerverről, majd a szerver ugyan ebben az üzenetben küldi vissza a lekérdezés eredményét.
- **DatabaseChangedMessage:** ez az üzenet azt jelzi, hogy az adatbázis tartalma megváltozott. A szerver küldi a kliensnek, aki ekkor egy GetRecordMessage-el megismétli az adatok lekérdezését.

- **ErrorMessage:** nem várt hibák esetén, ezzel az üzenettel jelez a szerver a kliensnek.
- **LoadMessage:** ebben az üzenetben küldi el a kliens a feldolgozandó szerződést a szervernek.
- **ProgressInfoMessage:** ez az üzenet arra szolgál, hogy a szerver a folyamatok állapotáról értesítse a klienst.
- **LoginMessage:** ezzel az üzenettel jelzi a kliens a felhasználó bejelentkezési szándékát, majd a szerver ugyanebben az üzenetben küldi vissza az autentikáció eredményét.

5.4 Adatbázis

Az adatbázis, amit használok egy lokális [Microsoft SQL Server](#) relációs adatbázis. Mivel a rendszernek robosztus adatmennyiségre is fel kell lennie készülve, jó választásnak bizonyult, tudván, hogy az MSSQL ipari környezetben is megállja a helyét. Ezen kívül Microsoft környezetben ésszerű választásnak tűnt a magas szintű támogatottság miatt.

SQL adatbázis mellett döntöttem, mert egy noSQL adatbázis nagy lekérdezés komplexitás esetén problémákba ütközhet, valamint az adatbázisunk sémája itt előre definiált a noSQL adatbázis sémája viszont nem az. Ugyanakkor erősen fontolgattam noSQL adatbázis használatát, mert lényegesen egyszerűbb lett volna az objektum modell elemek kezelése. A jelenlegi formában a modell bővítése egy kis extra munkát vesz igénybe, mivel módosításkor az adatbázis szerkezetét is hozzá kell igazítani.

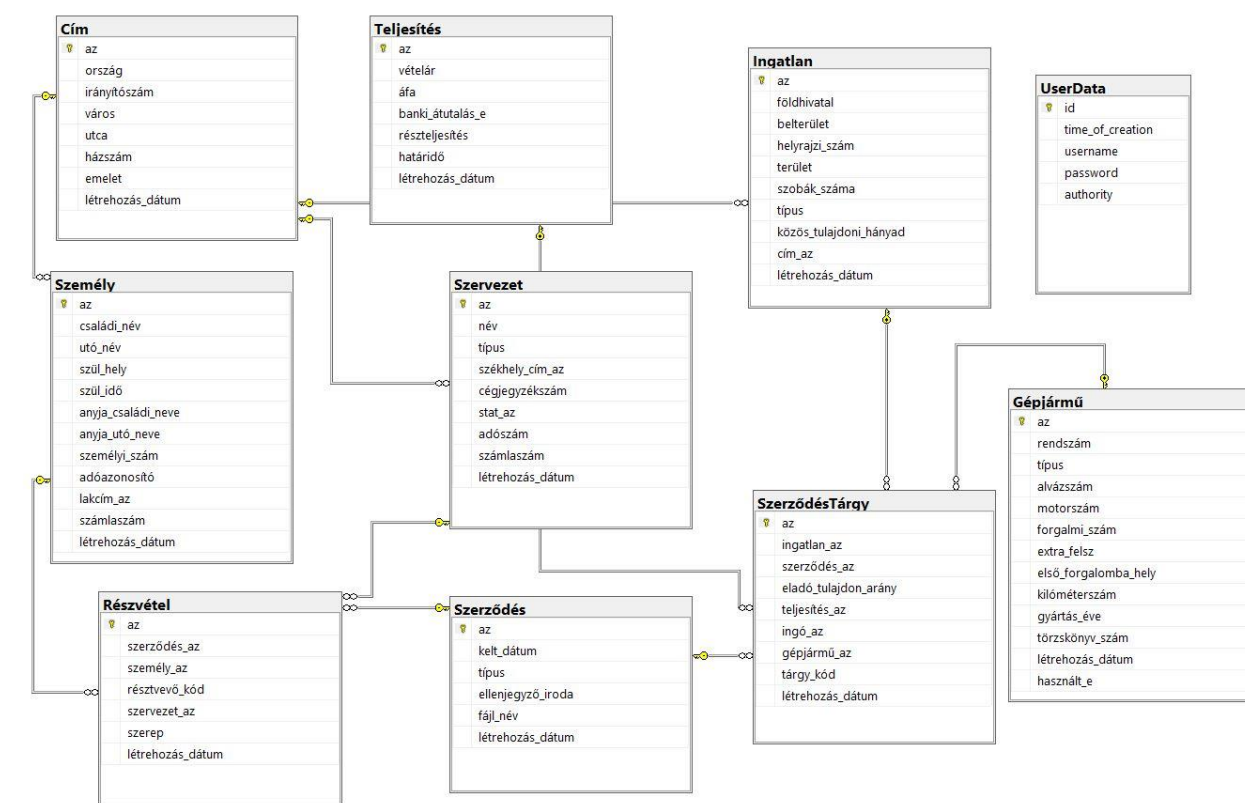
Az adatbázis sémája tartalmazza az összes, [Függelékben](#) is említett fontosabb attribútumot, ezeket a hovatartozásuknak megfelelő táblákba csoportosítva:

- UserData: a regisztrált felhasználók adatait tárolja autentikáció és autorizáció céljából.
- Cím: egy lakcímet, ingatlan címet vagy székhelyet tárol.
- Szerződés: egy szerződés általános információit tárolja.
- Személy: egy személy adatait tárolja.
- Szervezet: egy szervezet adatait tárolja.
- Teljesítés: egy szerződés teljesítésének feltételeit tárolja.
- Ingatlan: egy ingatlan adatait tárolja.

- A táblák között két kivétel van a SzerződésTárgy és a Részvétel. Ezek kapcsolattáblák, de saját tulajdonságokat is hordoznak:

- **SzerződésTárgy:** összekapcsolja a Szerződés táblát a Teljesítés és Ingatlan vagy Gépjármű táblákkal. Attól függően, hogy egy szerződésnek milyen fajta tárgya van. Ezen kívül itt rögzítjük az eladó tulajdonarányt.
- **Résztétel:** összekapcsolja a Szerződés táblát a Személy vagy Szervezet táblákkal, attól függően, hogy a résztvevő személy-e vagy szervezet. Ezen kívül tárol egy szerep nevű attribútumot, ami megadja, hogy eladó vagy vevő-e a résztvevő.

Az előbb említett adatbázis táblákat és azok kapcsolatát a **5.10. ábra** szemlélteti.



6 Összegzés

A munkám célja az volt, hogy létrehozzak egy alkalmazást, ami a tetszőleges adásvételi szerződéseket beolvasva kigyűjti és a felhasználó rendelkezésére bocsátja, a benne található fontosabb információkat. A szoftver asztali alkalmazás formájában valósult meg, egy szerverre és kliensre bontva, biztosítva a köztük lévő kommunikációt egy Tcp-Socket API-n keresztül. A munka során a természetes szöveg egyes entitásainak felismeréséhez saját algoritmust írtam, valamint a gépi tanuláson alapuló Named Entity Recognition-t hasznosítottam és a két módszer kombinációjaként állt elő a végleges szövegelemző. A NER csővezeték előállításához egy előre betanított modellt használtam, majd finomhangoltam. Mindkét módszer használata előtt elengedhetetlen volt elegendő méretű adathalmazok begyűjtése és az elvárt formátumra hozása.

A fejlesztés során rengeteg akadályba ütköztem, mivel az említett NLP technológiákkal a dolgozat elkészítésének keretein belül, de az akadályokat többnyire sikerült áthidalnom az eredeti ötlettől eltérő megközelítéseket alkalmazva. Ezáltal, úgy gondolom nagy mennyiségű, hasznos és korszerű tudásra tettem szert a témakörben.

6.1 Eredmények

Az a meggyőződésem, hogy az elkészült alkalmazás koránt sem teljes, rengeteg lehetőség van a bővítésére, javítására. A manuális tesztek során helyesen felismerte és összekapcsolta a legtöbb keresett információt a szövegekben, de nem mindig hozott jó döntést.

A rendszer képes volt felismerni, akár több eladó és vevő esetén is a résztvevőket és azok adatait, legyenek azok személyek, szervezetek vagy vegyesen.

Ezen kívül a szerződés tárgyát és annak adatait, ingatlan és gépjármű adásvételi szerződések esetén, de egyelőre csak egy konkrét tárgyra képes felismerni.

Valamint a teljesítés feltételeit is megtalálja, mint határidő, vételár stb. A program nem kezeli például a csereszerződéseket és az ingóságokra vonatkozó adásvételeket, de úgy lett megírva, hogy jól skálázható legyen nemcsak új típusú szerződésekre, de a meglévő entitások bővítésére, például egy új fajta résztvevő felvétele esetén vagy egy új ingatlan tulajdonság hozzáátételével, mint például, hogy tartozik-e garázs az ingatlanhoz vagy nem.

Továbbá rendszer alkalmas több felhasználó kérését kiszolgálni fennakadás nélkül egy hosszabb feldolgozás közben is.

Sajnos az automatikus teszteléshez még mindig kevés szerződés minta áll rendelkezésre, mivel a rendelkezésre állók adatait felhasználom a finomításhoz, így konkrét szám adatot a hatékonyságról még nem tudok felmutatni. Ez egy nyitott kérdés a jövőre nézve, hogy hogyan lehet vizsgálni, hogy a rendszer milyen pontossággal talál meg adatokat.

6.2 Továbbfejlesztési lehetőségek

Több tervem is van a jövőre nézve a rendszer fejlesztésére. Először is a szerveren szeretnék egy funkciót kifejleszteni, amivel a szövegfeldolgozás eredményének hatékonysága mérhető entitásonként és összegezve is. Ehhez további adatgyűjtések által szándékozok egy teszt adathalmazt felépíteni.

Továbbá szeretném kiterjeszteni a támogatott szerződéstípusokat, például csereszerződések bevételeivel. A keresett adatok listáját is bővíteném új attribútumokkal, valamint a már meglévők megtalálásának hatékonyságát javítanám további szerződésekből kinyert információkkal, az algoritmus finomításával és a NER modell újratanításával, olyan adathalmazzal, amely több fajta címkét is tartalmaz például gépjárművek azonosítására. A tanításhoz használnék majd nagyobb adathalmazt több GPU memória birtokában.

Ezen kívül tervben van, egy funkció, hogy egy felhasználó beolvashasson szerződéseket kizárólag saját felhasználásra és ezeket csak ő láthassa egy privát adatbázisban, a közös adatbázis mellett.

Ezen kívül tervben van a felhasználói interfész átmozgatása webes felületre, hogy a szolgáltatások könnyebben elérhetőek legyenek.

7 Irodalomjegyzék

- [1] 2013. évi V. törvény a Polgári Törvénykönyvről XXXII. Fejezet: Az adásvételi szerződés általános szabályai, URL: http://cic.ckh.hu/sites/default/files/jogihatter/ptk_2017.pdf (utoljára látogatva: 2021.09.11)
- [2] Python Software Foundation: *Python programozási nyelv*, URL: <https://www.python.org/>
- [3] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan: *PyTorch, open source machine learning library*, URL: <https://pytorch.org/docs/stable/index.html>
- [4] Clement Delangue and Julien Chaumond: *Hugging Face*, 2016, URL: <https://huggingface.co/>
- [5] Hugging Face: *transformers package*, an immensely popular Python library providing pretrained models URL: <https://huggingface.co/transformers/>
- [6] Wes McKinney and the Pandas Development Team: *pandas, powerful Python data analysis toolkit Release 1.3.4*, URL: <https://pandas.pydata.org/docs/pandas.pdf>
- [7] Ignace Maes: *Material Skin, Theming .NET WinForms, C# or VB.Net, to Google's Material Design Principles*, URL: <https://github.com/IgnaceMaes/MaterialSkin>
- [8] Joel Christner: *Simple wrapper for TCP client and server in C# with SSL support*, URL: <https://github.com/jchristn/SuperSimpleTcp>
- [9] Iron Software LLC: *IronOcr for .NET, The C# OCR Library*, URL: <https://ironsoftware.com/csharp/ocr/>
- [10] Hewlett-Packard: *Tesseract, open source optical character recognition engine for various operating systems*, URL: <https://github.com/tesseract-ocr/tesseract>
- [11] *Microsoft Language Integrated Query (LINQ) (C#)* URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>
- [12] *In a recent survey conducted by DBmaestro, MS SQL lead the pack among respondents.* URL: <https://dzone.com/articles/6-reasons-why-microsoft-sql-is-alive-and-well>
- [13] Jacob Devlin Ming-Wei Chang Kenton Lee Kristina Toutanova, Google AI Language: *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding 2019*, URL: <https://arxiv.org/pdf/1810.04805.pdf> (utoljára látogatva: 2021.10.03)

- [14] Jogkódex - Internetes jogi tudástár URL: https://hvgorac.hu/jogkodex_csoport (utoljára látogatva: 2021.10.23)
- [15] Szeged Treebank György Szarvas, Richárd Farkas, László Felföldi, András Kocsor, János Csirik: *Highly accurate Named Entity corpus for Hungarian. International Conference on Language Resources and Evaluation 2006, Genova (Italy)*, URL: <https://rgai.inf.u-szeged.hu/node/130> (utoljára látogatva: 2021.12.21)
- [16] Rahimi, Afshin and Li, Yuan and Cohn, Trevor: *Wikiann Massively Multilingual Transfer for {NER} - Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, URL: <https://www.aclweb.org/anthology/P19-1015>
- [17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez†, Łukasz Kaiser, Illia Polosukhin‡: *Attention Is All You Need* 2017, URL: <https://arxiv.org/pdf/1706.03762.pdf> (utoljára látogatva: 2021.12.05)
- [18] jacobdevlin-google: *bert-base-multilingual-cased*, URL: <https://github.com/google-research/bert/blob/master/multilingual.md>
- [19] it.uu.se: *Sockets API*, URL: <https://www.it.uu.se/education/course/homepage/dsp/vt19/modules/module-2/sockets/> (utoljára látogatva: 2021.12.05)
- [20] by Makoto Hiramatsu of Cookpad Inc.: blog post about *nerman: Named Entity Recognition System Built on AllenNLP and Optuna*, URL: <https://medium.com/optuna/nerman-named-entity-recognition-system-built-on-allennlp-and-optuna-c044c319b955> (utoljára látogatva: 2021.10.02)
- [21] Nemeskey, Dávid Márk (2020): *"Natural Language Processing Methods for Language Modeling."* PhD Thesis. Eötvös Loránd University. URL: https://hlt.bme.hu/en/publ/nemeskey_2020 (utoljára látogatva: 2021.09.05)
- [22] Nemeskey, Dávid Márk (2021): *"Introducing huBERT."* In: XVII. Magyar Számítógépes Nyelvészeti Konferencia (MSZNY 2021). Szeged, pp. 3-14. URL: https://hlt.bme.hu/en/publ/hubert_2021 (utoljára látogatva: 2021.09.05)
- [23] Pablo Sánchez: *Different methods for mitigating overfitting on Neural Networks*, URL, <https://quantdare.com/mitigating-overfitting-neural-networks/> (utoljára látogatva: 2021.12.07)

8 Függelék

8.1 Fontosabb attribútumok

Azoknak az attribútumoknak a gyűjteménye, melyeket a program keres egy adásvételi szerződésben:

- Címek: emelet, házszám, irányítószám, ország, utca, város.
- Gépjárművek: alvázszám, első forgalomba helyezés ideje, extra felszereltségek, forgalmi szám, gyártás éve, használt / nem használt, kilométerszám, motorszám, rendszám, típus, törzskönyvszám.
- Ingatlanok: belterület, földhivatal, helyrajzi szám, ingatlantípus, közös tulajdoni hányad, szobák száma, terület, cím.
- Személyek: adószám, anyja neve, családi név, utónév, számlaszám, személyi szám, születési dátum, lakcím, születési hely.
- Szervezetek: adószám, cégjegyzékszám, cégnév, cégtípus, statisztikai azonosító, számlaszám, székhely.
- Szerződés általános információk: eladó tulajdonarány, keltezés, közjegyző iroda, szerződéstípus.
- A teljesítésre vonatkozó információk: áfa, átutalás-e, határidő, részletek, vételár.

8.2 Az elkészült projekt

A projekt fájlok együttes mérete még tömörítve is meghaladja a mellékletként feltölthető megengedett 15MB-ot, ezért a munkám egy Google Drive linken keresztül osztom meg:

<https://drive.google.com/drive/folders/1hlSvxnGY3aZiivkcqW3GyN0hC9Cky4-v?usp=sharing>

A fejlesztés közben GitHub-ot használtam git szolgáltatóként, ezért alap esetben azon keresztül osztanám meg a projektet, de az nem foglalná magába a NER modellt, amit az alkalmazás használ futás közben, mivel annak mérete önmagában meghaladja a 450 MB-ot és ekkora fájl nem tölthető fel GitHub-ra.

Ezen kívül a betanított NER modell elérhető és kipróbálható kisebb szövegekkel az alábbi Hugging Face linken: <https://huggingface.co/fdominik98/ner-hu-model-2021>