Az információk a programomhoz használt filmekről a „movies.db” adatbázisban kerültek letárolásra. Az adatbázishoz SQLite formátumot használtam annak egyszerűsége, hordozhatósága miatt, illetve egy nagy előnye, hogy megbízhatóan tud működni gyakorlatilag szinte minden programozási környezetben.

A felhasznált adatok egy kaggle.com-ról származó adatbázisból származnak, melynek forrása a következő: <https://www.kaggle.com/datasets/tmdb/tmdb-movie-metadata>

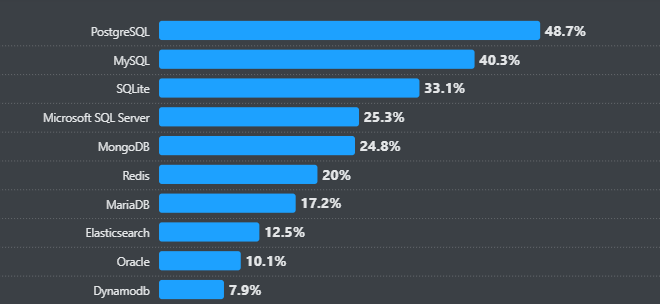
A felhasznált adatbázis a Tmdb-ről származó adatokkal lett feltöltve, így az általam készített adatbázis adatainak az eredeti forrása a Tmdb-n szereplő adatok.

**SQLite bemutatása**

Az SQLite egy olyan nyílt forráskódú C nyelvű könyvtár, amely egy cross-platform, teljes körű SQL adatbázis-motort valósít meg. Jellemzői közé tartozik, hogy rendkívül kicsi, gyors, önálló, és nagy megbízhatóságú.[[1]](#footnote-1)

Az SQLite a világ egyik leggyakrabban használt adatbázis-motorja.1 A cross-platform tulajdonságának köszönhetően számos különböző eszközön, köztük mobiltelefonokban, televíziókban, játékkonzolokban, kamerákban, autókban, és a legtöbb számítógépben is megtalálható valamilyen formában, vagy beépítve, vagy pedig olyan alkalmazások részeként, amelyeket az emberek nap mint nap használnak ezeken az eszközökön. [[2]](#footnote-2)

A StackOverflow 2024-es felmérése[[3]](#footnote-3) alapján a válaszokból ítélve a felhasználók 33,1%-a válaszolta azt, hogy használja az SQLite-ot, ezzel a listán a 3. helyet szerezte meg a legnépszerűbb, legelterjedtebb adatbázisok között.



1.1 ábra: A StackOverflow 2024-es felmérése különböző adatbázis motorok elterjedtségéről 3

A listában csak a PostgreSQL és a MySQL előzik meg, viszont ezek nehezen tekinthetőek az SQLite konkurenseinek, mivel ezek kliens/szerver adatbázis motorok, amelyek más jellegű problémákra nyújtanak megoldást. Általánosságban elmondható, hogy ezeknél az elosztott vállalati adatoknak egy központosított adatbázisban való tárolása a cél. Ezzel szemben az SQLite arra a célra készült, hogy helyi tárolás céljából biztosítson egy kisebb méretű adatbázist különböző alkalmazásokhoz, programokhoz, amelyben egyszerűen és gyorsan tárolhatók azok az adatok, amelyekkel a program dolgozik.2

Mivel az én szakdolgozatom középpontjában egy helyi, C# program áll, így a kliens/szerver adatbázis motorok helyett egy kisebb méretű helyi adatbázira volt szükségem, ezért is esett a választásom az SQLite-ra.

Ettől eltekintve az SQLite nem csak helyi programok esetében használható, sőt kliens/szerver adatbázismotort igénylő feladatok is megvalósíthatóak vele, vagy akár egy közepes látogatottságú weboldal adatbázisaként is használható.2

Mivel egy C# nyelvű programról van szó, ezért az adatbázis motor kiválasztásánál egy fontos kitétel volt, hogy C# nyelvvel könnyen, és jól használható legyen. A System.Data.SQLite egy ADO.NET szolgáltatás az SQLite-hoz.[[4]](#footnote-4) Az SQLite adatbázis motor ennek a System.Data.SQLite NuGet csomagnak a segítségével könnyedén használható, és integrálható bármely C# alkalmazásba, így tökéletes választásnak bizonyult egy SQLite adatbázis a szakdolgozatom programjához szükséges adatok eltárolásához.

**Az adatbázis szerkezete**

Az adatbázis 11 darab táblát tartalmaz, amelyek a következőek: Movies, Countries, Directors, Genres, Keywords, Languages, Movies\_Countries, Movies\_Directors, Movies\_Genres, Movies\_Keywords, Movies\_Languages.

A **Movies** tábla a következő mezőkből áll:

* ID:
  + Típus: INTEGER
  + Leírás: A film egyedi azonosítója, ez a primary key (elsődleges kulcs).
* Title:
  + Típus: TEXT
  + Leírás: A film címe angolul, amely az esetek nagy részében megegyezik a filmek eredeti címével.
* Genre:
  + Típus: TEXT
  + Leírás: A film műfaja(i) felsorolás szerűen, vesszővel elválasztva. Egyes filmeknél csak 1 darab van belőle, más, több műfajba is beleillő filmeknél több is lehet.
* Released:
  + Típus: INTEGER
  + Leírás: A film megjelenésének éve.
* Runtime:
  + Típus: INTEGER
  + Leírás: A film játékideje percben megadva.
* Gender\_of\_the\_protagonist:
  + Típus: INTEGER
  + Leírás: A főszereplő nevét jelzi. Az értéke 1, ha a főszereplő nő, ha férfi akkor pedig 2 az értéke.
* Main\_actor:
  + Típus: TEXT
  + Leírás: A főszereplő színész neve.
* Keywords:
  + Típus: TEXT
  + Leírás: Az adott filmhez kapcsolódó, a Tmdb-ről származó kulcsszavak vesszővel ellátott felsorolása, melyek használatával könnyebben találhatunk hasonlóságokat a filmek között.
* Director:
  + Típus: TEXT
  + Leírás: Az adott film rendezőjének/rendezőinek neve.
* Language:
  + Típus: TEXT
  + Leírás: A film eredeti nyelve, illetve a benne elhangzó nyelvek felsorolása
* Pruduction\_countries:
  + Típus: TEXT
  + Leírás: A készítő országok neveinek felsorolása.
* Tmdb\_score:
  + Típus: REAL
  + Leírás: Az adott film Tmdb-n szereplő értékelése, amely a felhasználói értékelések átlaga. Az értékelés egy 0-tól 10-ig terjedő skálán történik, így a pontszám is a két érték közé eshet csak.
* Number\_of\_ratings:
  + Típus: INTEGER
  + Leírás: Megmutatja, hogy a Tmdb oldalán eddig hányan értékelték az adott filmet, tehát a „Tmdb\_score” nevű mező hány értékelésnek az átlaga.
* Popularity:
  + Típus: REAL
  + Leírás: Az adott film Tmdb szerinti népszerűségi indexe, melyet a felhasználók tevékenységeiből számolnak.
* Budget:
  + Típus: INTEGER
  + Leírás: Az adott film költségvetésének összege amerikai dollárban megadva.
* Revenue:
  + Típus: INTEGER
  + Leírás: Az adott film bevételének összege amerikai dollárban megadva.

A **Countries** **tábla** a következő mezőkből áll:

* ID:
  + Típus: INTEGER
  + Leírás: Az adott ország egyedi azonosítója, ez a primary key.
* Country\_Name:
  + Típus: TEXT
  + Leírás: Az adott ország neve.

A **Directors** **tábla** a következő mezőkből áll:

* ID:
  + Típus: INTEGER
  + Leírás: Az adott rendező egyedi azonosítója, ez a primary key.
* Director\_Name:
  + Típus: TEXT
  + Leírás: Az adott rendező neve.

A **Genres** **tábla** a következő mezőkből áll:

* ID:
  + Típus: INTEGER
  + Leírás: Az adott műfaj egyedi azonosítója, ez a primary key.
* Genre\_Name:
  + Típus: TEXT
  + Leírás: Az adott műfaj neve.

A **Keywords** **tábla** a következő mezőkből áll:

* ID:
  + Típus: INTEGER
  + Leírás: Az adott kulcsszó egyedi azonosítója, ez a primary key.
* Keyword\_Name:
  + Típus: TEXT
  + Leírás: Az adott kulcsszó neve

A **Languages** **tábla** a következő mezőkből áll:

* ID:
  + Típus: INTEGER
  + Leírás: Az adott nyelv egyedi azonosítója, ez a primary key.
* Language\_Name:
  + Típus: TEXT
  + Leírás: Az adott nyelv elnevezése

A **Movies\_Countries** **kapcsolótábla** a következő mezőkből áll:

* Movie\_ID:
  + Típus: INTEGER
  + Leírás: Az adott film egyedi azonosítója, foreign key, amely a Movies tábla ID mezőjére mutat.
* Countries\_ID:
  + Típus: INTEGER
  + Leírás: Az adott ország egyedi azonosítója, foreign key, amely a Country tábla ID mezőjére mutat.

A **Movies\_Directors** **kapcsolótábla** a következő mezőkből áll:

* Movie\_ID:
  + Típus: INTEGER
  + Leírás: Az adott film egyedi azonosítója, foreign key, amely a Movies tábla ID mezőjére mutat.
* Directors\_ID:
  + Típus: INTEGER
  + Leírás: Az adott rendező egyedi azonosítója, foreign key, amely a Director tábla ID mezőjére mutat.

A **Movies\_Genres** **kapcsolótábla** a következő mezőkből áll:

* Movie\_ID:
  + Típus: INTEGER
  + Leírás: Az adott film egyedi azonosítója, foreign key, amely a Movies tábla ID mezőjére mutat.
* Genres\_ID:
  + Típus: INTEGER
  + Leírás: Az adott műfaj egyedi azonosítója, foreign key, amely a Genre tábla ID mezőjére mutat.

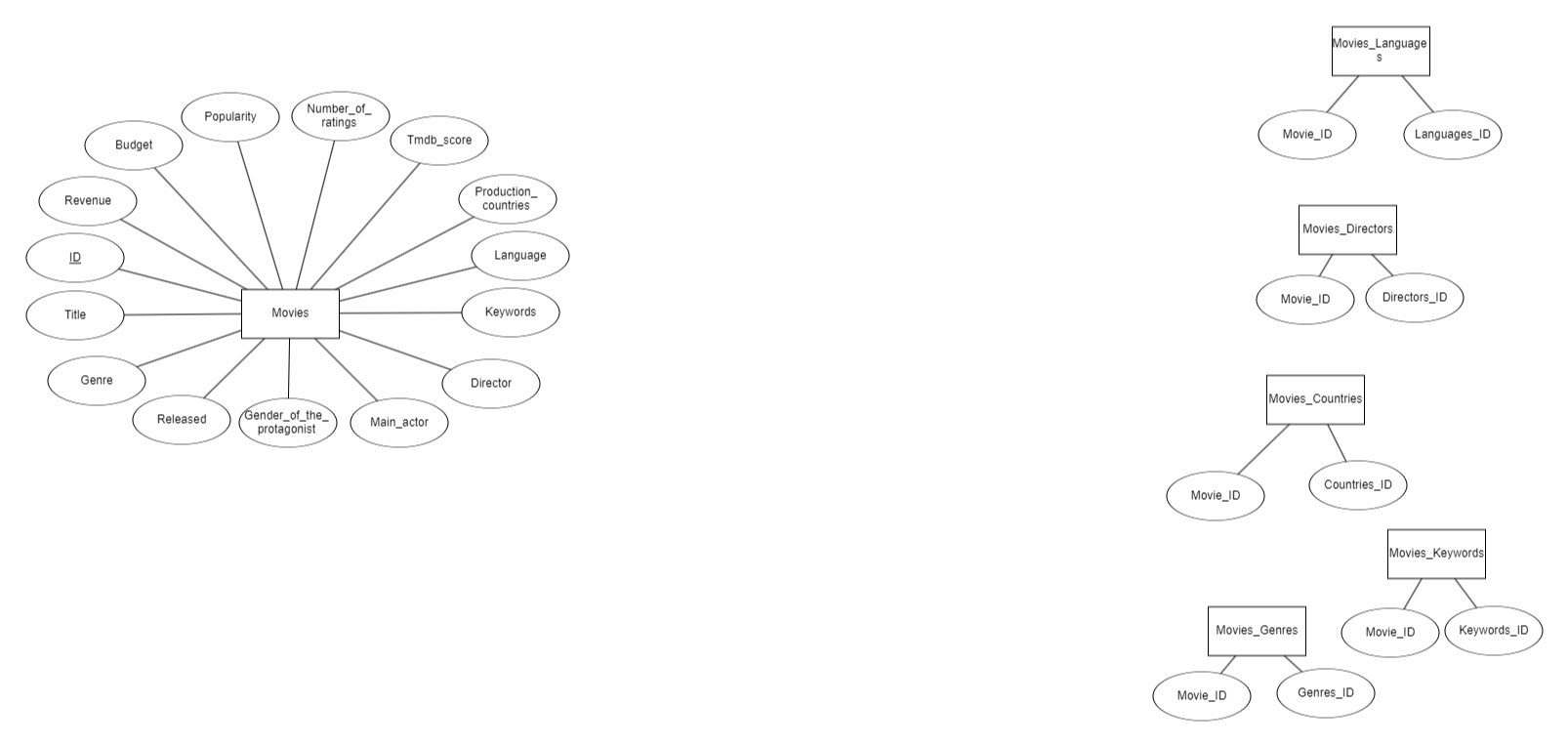
A **Movies\_Keywords** **kapcsolótábla** a következő mezőkből áll:

* Movie\_ID:
  + Típus: INTEGER
  + Leírás: Az adott film egyedi azonosítója, foreign key, amely a Movies tábla ID mezőjére mutat.
* Keywords\_ID:
  + Típus: INTEGER
  + Leírás: Az adott kulcsszó egyedi azonosítója, foreign key, amely a Keyword tábla ID mezőjére mutat.

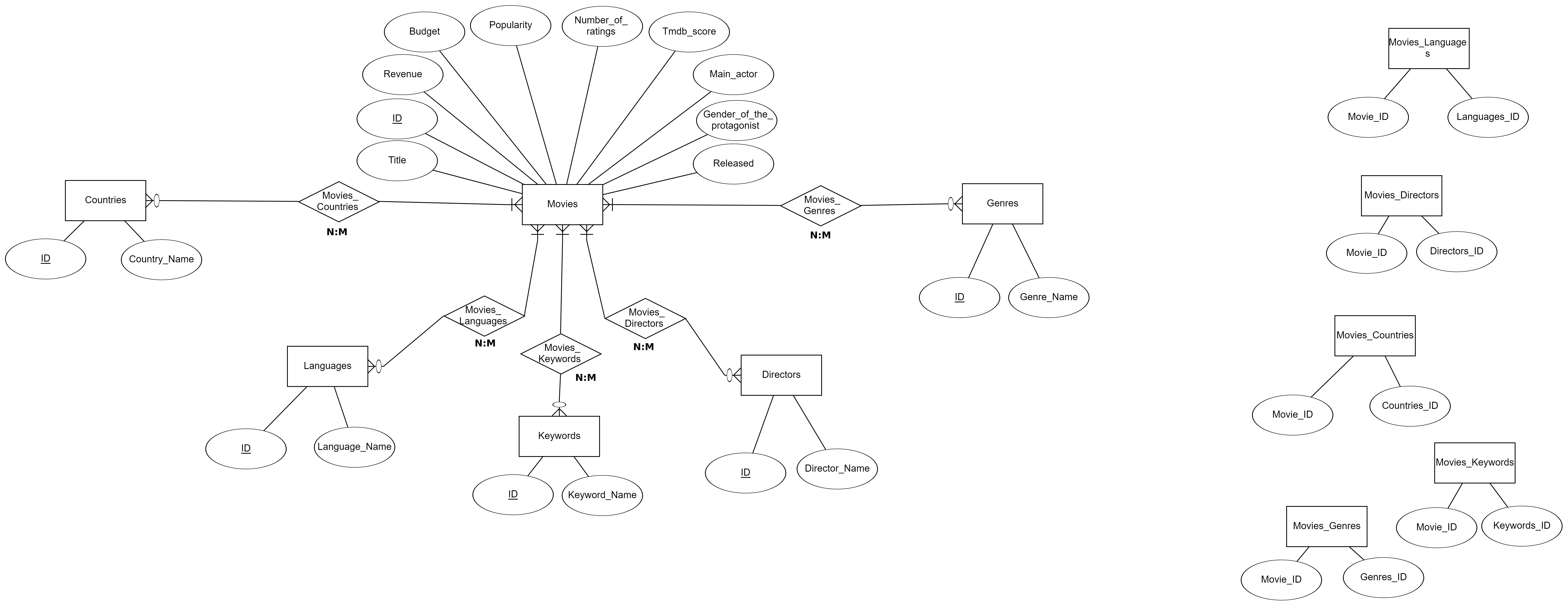
A **Movies\_Languages** **kapcsolótábla** a következő mezőkből áll:

* Movie\_ID:
  + Típus: INTEGER
  + Leírás: Az adott film egyedi azonosítója, foreign key, amely a Movies tábla ID mezőjére mutat.
* Languages\_ID:
  + Típus: INTEGER
  + Leírás: Az adott nyelv egyedi azonosítója, foreign key, amely a Language tábla ID mezőjére mutat.

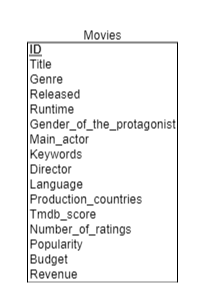
**Az adatbázis ER modellje lebontás előtt**

****

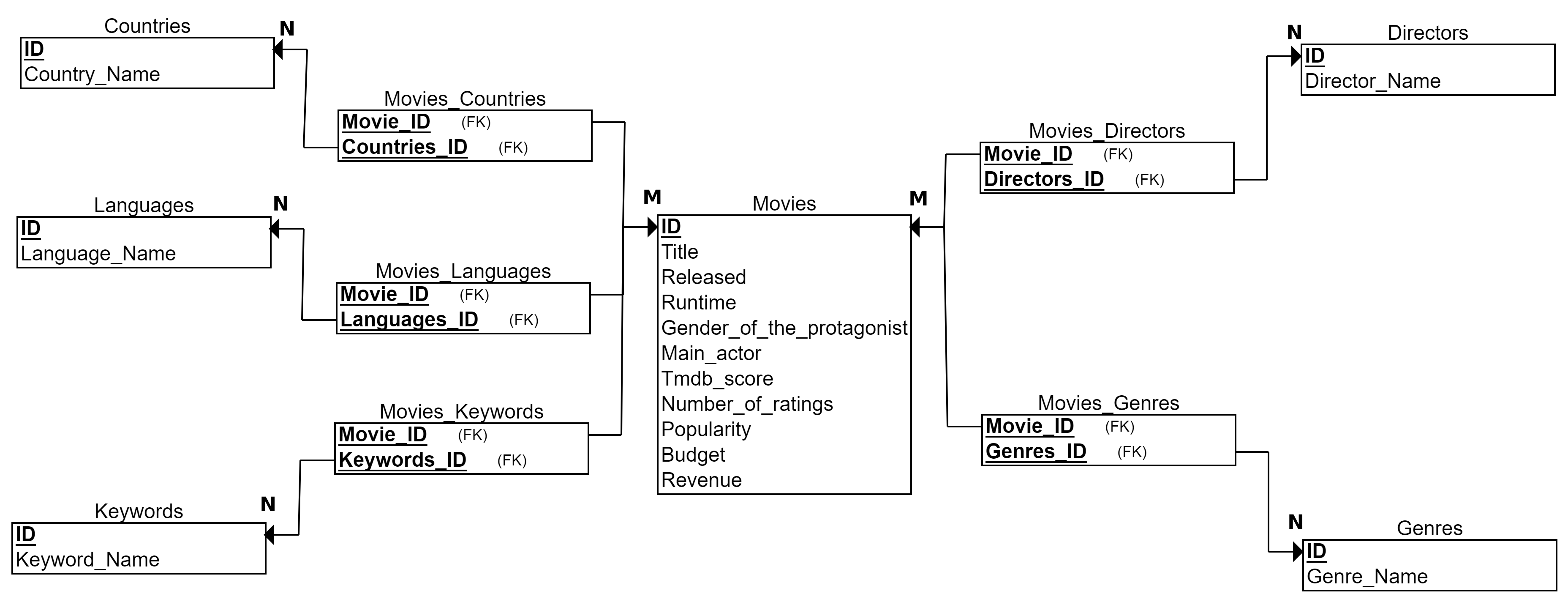
**Az adatbázis ER modellje lebontás után**

****

**Az adatbázis relációs modellje lebontás előtt**

****

**Az adatbázis relációs modellje lebontás után**

****

A lebontás során 1 táblából 11 táblánk lesz. Minden lebontott mezőhöz 1 tábla és egy kapcsolótábla is fog tartozni.

Minden lebontott tábla N:M kapcsolatot valósít meg a Movies táblával, hiszen pl. egy filmnek lehet több műfaja, illetve ha van egy műfaj, akkor az több filmnél is szerepelhet, nem csak egy darabnál. Egy filmben beszélhetnek több nyelven is, továbbá több filmben is beszélhetnek ugyanolyan nyelven, így ugyanazt a nyelvet több filmhez is hozzárendelhetjük, és így tovább. Az N:M kapcsolat miatt van tehát szükség a kapcsolótáblákra minden lebontott mező esetében.

A táblák közötti kapcsolatok egyik irányban kötelezőek, másik irányban viszont opcionálisak: Minden filmnek van műfaja, gyártási országa, rendezője, stb., ezért ez kötelező jellegű kapcsolat. Viszont belekerülhetnek az adatbázisba olyan műfajok, országok, rendezők, amiknél előfordulhat, hogy nincs olyan film az adatbázisban amik ilyen tulajdonságokkal rendelkeznek, ezért ezekhez nem biztos hogy lesznek rendelve filmek, így ez opcionális jellegű kapcsolat.

**Kézzel készített döntési fa a film nyelvére vonatkozóan**

A fa kézzel került összerakásra, a leírásához JSON formátumot választottam. Tanító halmaznak egy 20 elemű halmazt vettem, ami az adatbázisom utolsó 20 db filmjéből áll, azok alapján alakítottam ki az elágazásokat.

A JSON fájl felépítésében 3 ismétlődő elem jelenik meg: a „node”, amely a csomópontokat jelöli, a „condition”, ami a feltételeket, elágazásokat jelöli, illetve az „outcome”, amelyek a fa levelei, tehát a lehetséges kimenetek.

Az alábbi kódrészlet a döntési fa egyik ága, amelyet akkor vizsgálunk, ha az adott film Tmdb értékelése 6.0, vagy annál kisebb. Ezután a következő csomópont a Budget: ha nagyobb, mint 70000000, akkor a film nyelve angol. Ha kisebb, vagy egyenlő, mint 70000000, akkor még a film Popularity tulajdonságának vizsgálatára is szükség van. Ha ez az érték nagyobb, mint 20, akkor a film nyelve, tehát a fa kimenete Angol és Spanyol, ellenben ha kisebb, vagy egyenlő mint 20, akkor Francia.

{

"decision\_tree": {

"root": {

"node": "Tmdb-score",

"condition": [

{

"condition": "<= 6.0",

"outcome": {

"node": "Budget",

"condition": [

{

"condition": ">70000000",

"outcome": "Angol"

},

{

"condition": "<=70000000",

"outcome": {

"node": "Popularity",

"condition": [

{

"condition": ">20",

"outcome": ["Angol", "Spanyol"]

},

{

"condition": "<=20",

"outcome": "Francia"

}

]

}

}

]

}

}

**Kézzel készített döntési fa hatékonysága**

A bemutatott döntési fa hatékonyságának mérését a következő táblázat mutatja be.

|  |  |  |  |
| --- | --- | --- | --- |
| Intervallum | "1-20" | "60-80" | Összesen |
| Helyes | 17 | 4 | 21 |
| Összesen | 20 | 20 | 40 |
| Hatékonyság | 85,00% | 20,00% | 52,50% |

**1**. táblázat: A kézzel készített döntési fa hatékonyságának mérési eredményei

A fa hatékonyságának mérését a movies.db adatbázis két különböző, de egyforma méretű tartományán végeztem. A mérésnél az egyes eseteknél abban az esetben soroltam „Helyes” kategóriába az adott esetet, ha a film nyelveit a különbőző tulajdonságainak vizsgálatával, a döntési fát használva pontosan sikerült megállapítani. A többnyelvű filmeknél, pl. a "Magyar, Angol"-t egy nyelvnek vettem, hiszen a döntési fa kialakításánál is ahol több nyelv volt jelen egy filmben, azokat együttesen vettem egy nyelvként. Tehát pl. ha egy film nyelve „Angol, Spanyol”, de a döntési fát használva végeredményként amennyiben csak az „Angol”, vagy esetleg csak a „Spanyol” eredmény jött ki végeredménynek, akkor az adott filmre vonatkozó becslés nem bizonyult „Helyes” eredményűnek.

Az 2.-es táblázat eredményeiből jól látszik, hogy a hatékonyság nagyban függ attól, hogy az adatbázis melyik tartományában vizsgáljuk a filmeket. Hatékonyságra az első 20 db filmet vizsgálva 85% jött ki eredményként, mivel az adatbázis ezen szakaszán nagyrészt angol nyelvű filmek voltak csak, így itt jobban működött a fa. A 61-80. filmre is megnéztem a hatékonyságát, itt láthatóan kevésbé működött jól a fa, csak 20%-os hatékonyságot adott, ami azzal magyarázható, hogy ebben az intervallumban sokkal vegyesebb nyelvű filmek szerepelnek, illetve sok olyan nyelvkombináció (pl. "Afrikai, Angol", "Angol, Japán") jelent meg, amelyek a döntési fában nem szerepeltek, hiszen az utolsó 20 filmnél nem voltak ezek jelen. A két szeletet együttvéve, a 40 filmet összességében vizsgálva 52,5%-os hatékonyság jön ki eredményként.

Később a programom DecTreeForLanguage osztályában létrehoztam egy CompareToHandmadeDecTree metódust, amelyben az ML.NET segítségével létrehoztam egy hasonló becslést, mint a kézzel készített döntési fa. A metódus célja, mint a neve is jelöli, az volt, hogy össze lehessen hasonlítani egy gép által létrehozott becslést a kézzel készített döntési fával. Ezt úgy valósítottam meg, hogy a filmek ugyanazon jellemzőit adtam meg betanítási adatnak, mint amik a kézzel készített fában is szerepelnek, tehát a Budget, Popularity, NumberOfRatings, GenderOfProtagonist és a TmdbScore jellemzőket. A kimenet, tehát a jellemző, amire a becslés irányult a film nyelve volt értelemszerűen itt is. Betanításhoz ugyanúgy az adathalmaz utolsó 20 filmjét használtam fel, mivel ezek alapján készült ugye a kézzel készített fa is. A hatékonyság vizsgálatakor ugyanúgy 2 tartományt vizsgáltam: az első 20 filmet, illetve a 60. és 80. film közti intervallumban lévő filmeket.

A kapott eredményeket a következő táblázat tartalmazza:

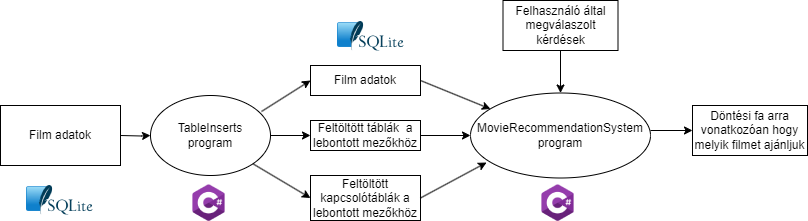
|  |  |  |  |
| --- | --- | --- | --- |
| Intervallum | "1-20" | "60-80" | Összesen |
| Helyes | 17 | 7 | 24 |
| Összesen | 20 | 20 | 40 |
| Hatékonyság | 85,00% | 35,00% | 60,00% |

**2**. táblázat: Az ML.NET-tel készített becslő algoritmus hatékonyságának mérési adatai

A mérés módja és szempontjai teljes mértékben megegyeztek a korábban a fa mérésénél ismertetett módszerrel, csak jelen esetben automatizálva történtek. A CompareToHandmadeDecTree metódusban a becsült értékeket összevetettem a valós értékekkel, és ha egyezést találtam, akkor egy erre a célra létrehozott double típusú változó értékét megnöveltem. Miután végigment a program a filmeken, a kapott értéket elosztottam 20-szal, mivel egy 20 elemű intervallumot vizsgáltunk, így megkaptam a hatékonyságot százalékos formában.

A mérési adatok összehasonlításából az látszik, hogy az első vizsgált intervallumban ugyanolyan hatékonysággal tudott működni az algoritmus, mint a kézzel készített fa, tehát 85%-os hatékonysággal. A második intervallumban, amely vegyesebb nyelvű filmeket tartalmaz, viszont tudott javulni a kézzel készített becsléshez képest, mivel a 20 filmből 4 helyett 7 darabot sikerült helyesen megbecsülnie, ezzel pedig a teljes hatékonyságot 52,5%-ról 60%-ra sikerült növelni. Tehát ezek alapján a géppel készített becslő algoritmus 7,5%-kal hatékonyabb, mint a kézzel készített döntési fa.

**Folyamatábra az adatbázis adatainak feldolgozásáról**

****

**ML.NET**

A mesterséges intelligencia egy olyan ága a számítástechnikának, amely azzal foglalkozik, hogy a számítógépeket olyan feladatok elvégzésére tanítja be, amelyekhez általában valamilyen emberi beavatkozás, emberi intelligencia szükséges. Napjainkban megkérdőjelezhetetlen ütemben fejlődik és terjed, az élet egyre több területén jelenik meg a használata, és folyamatosan beépítésre kerül a különböző szolgáltatásokba, rendszerekbe. A mesterséges intelligenciának több ágazata is van, ezek közül egyik meghatározó ága a gépi tanulás. [[5]](#footnote-5)

A gépi tanulás alatt egy olyan folyamatot értünk, amely automatikusan képessé tesz egy gépet vagy rendszert arra, hogy tanuljon és javuljon a feldolgozott adatokból vett tapasztalatokból, ezáltal előrejelzéseket, becsléseket tud készíteni az új adatokra vonatkozóan. Tehát az explicit programozás helyett a gépi tanulás algoritmusokat használ nagy mennyiségű adatok elemzésére, a felismerésekből való tanulásra, majd ezek alapján a döntések meghozatalára. 1

A gépi tanulási algoritmusok teljesítménye idővel javulni tud azáltal, hogy egyre több adattal találkoznak amelyeket elemeznek, majd mintákat ismernek fel belőle amiknek a megtanulásával egyre pontosabb eredményeket tudnak elérni.1

Maga a gépi tanulási modell alatt azt értjük, amit a program az algoritmus a tanító adatokon történő futtatásából tanul. Minél több adatot használunk, annál jobb, pontosabb lesz a modell. 1

A térhódításnak megfelelően a különböző programozási nyelvekhez is megjelentek az utóbbi években a különböző kiegészítők, függvénykönyvtárak, amelyek lehetővé teszik a már előre implementált gépi tanulást használó programrészek, algoritmusok beépítését a programjainkba.

A legelterjedtebb programozási nyelvek közül párat említve a c++-hoz az mlpack, a pythonhoz a scikit-learn függvénykönyvtárak használhatóak a gépi tanulás algoritmusainak eléréséhez, az általam is használt C# nyelv esetén pedig az ML.NET függvénykönyvtár tartalmazza ezeket az algoritmusokat.

Mi az az ML.NET?

Az ML.NET egy ingyenes, nyílt forráskódú és cross-platform, azaz Windows, Linux és macOS operációs rendszerekre is egyaránt elérhető gépi tanulási keretrendszer a .NET fejlesztői platformhoz. A keretrendszer nem csak C#, hanem a kevésbé ismert F# nyelvvel is kompatibilis.[[6]](#footnote-6) Az ML.NET segítségével betanítható egy egyéni modell, vagy akár importálhatóak előre betanított TensorFlow- és ONNX-modellek.[[7]](#footnote-7)

Az ML.NET tökéletesen használható számos kategorizálási, előrejelzési, összehasonlítási feladatra, többek között például az ügyfelek visszajelzéseinek kategorizálására (jó vagy rossz), folytonos jellemzők, tehát például termékek árának megbecslésére, dolgok hasonlóságának összehasonlítására, vagy akár a felhasználók korábbi tevékenységei, érdeklődési körei alapján történő ajánlások készítésére, ami az én szakdolgozatomnak is a feladata.3

A függvénykönyvtár első verziójának kiadására 2018 május 7-én került sor, a stabil kiadás legelső, 1.0-ás verzióját pedig 1 évvel később, 2019-ben a Microsoft Build nevű konferenciáján jelentették be.2 A szakdolgozatom írásakor a legfrissebb kiadott stabil verzió az a 2024 január 18-án megjelent 3.0.1-es verzió, így a program megírásához is ezt a verziót használtam.

Hogyan néz ki egy ML.NET alkalmazás?

Egy ML.NET program felépítése általánosságban a következőképpen néz ki:

* MLContext létrehozása

Egy ML.NET alkalmazás legelső lépése az MLContext létrehozása, hiszen ezek után kezdődhet csak el az adatok betöltése, átalakítása, a modell betanítása, az algoritmusok kiválasztása. Tehát az MLContext az egész ML.NET alkalmazás kiindulópontja.[[8]](#footnote-8)

var mlContext = new MLContext();

(Az ML.NET dokumentációjából4 származó kódrészlet)

* Adatok betöltése

Mint már korábban is említésre került, a gépi tanulás már ismert adatokat dolgoz fel, ezekben keres mintákat, majd az ezekből nyert ismeretek segítségével becsül meg ismeretlen adatokat, információkat. Tehát az ML.NET-nek is szüksége van ezekre az úgynevezett betanító adatokra, amiket fel tud használni a későbbi becslések során.

A bemeneti adatokat, amikben a minták keresése, felismerése történik, Feature-öknek nevezzük, így jelennek meg a kódban. A kimeneti adatok, amelyeknek az előrejelzése gyakorlatilag a programnak a célja, Label-ként szerepelnek a kódban. 4

Az adatok IDataView objektumokba kerülnek betöltésre, amelyek tartalmazhatnak különböző típusú számokat, stringeket, bool értékeket, vagy akár vektorokat és számokat tartalmazó tömbőket is tölthetünk bele, listákat viszont nem támogat. Az adatok betöltése, amennyiben fájlból történik, lehetséges többek között txt, csv, tsv és egyéb formátumokból is, viszont élő, valós idejű adatok betöltésére is lehetőség van. 4

IDataView trainingData = mlContext.Data.LoadFromTextFile<SentimentInput>(dataPath, separatorChar: ',', hasHeader: true);

(Az ML.NET dokumentációjából4 származó kódrészlet)

Amennyiben már memóriában lévő, változókból, tömbökből származó adatokat szeretnénk betölteni, akkor arra is lehetőségünk van.4

IDataView trainingData = mlContext.Data.LoadFromEnumerable<SentimentInput>(inMemoryCollection);(Az ML.NET dokumentációjából4 származó kódrészlet)

* Adatok átalakítása

Sok esetben az adatok hiába tartalmaznak rengeteg hasznos és értékes információt, eredeti formájukban, amikben betöltésre kerültek sajnos nem használhatóak, így átalakításokat kell végeznünk rajtuk annak érdekében, hogy a modellünk betanítására fel tudjuk őket használni.4 A string értékeket például számokká kell átalakítanunk, a különböző hosszúságú tömböket pedig ahhoz, hogy összehasonlíthatóak legyenek, elő kell készítenünk olyan módon, hogy egységes elemszámúak legyenek.

Számos lehetőséget kínál a függvénykönyvtár az adatok átalakítására. Az átalakításokat végző metódusok egy részének feltétele az, hogy csak a betanítási adatokra lehet őket meghívni, egyes adatátalakításokhoz viszont nem szükségesek a betanítási adatok, így használhatóak azok nélkül is.[[9]](#footnote-9) A teljesség igénye nélkül néhány, amelyek gyakoribb problémákat oldanak meg:

* ConvertType: A bemeneti oszlop típusát lehet vele átalakítani
* MapValueToKey: A művelet során a bemeneti adatok értékeit egy adott kategóriákhoz (kulcsokhoz) rendeljük, tehát minden kategória egyedi kulcsértékkel fog rendelkezni, amelyek alapján azonosítani tudjuk majd őket.
* TokenIntoWords: Egy vagy több darab, szöveget tartalmazó oszlop osztható fel vele szavakra
* OneHotEncoding: segítségével a kategóriákat különálló, bináris vektorokba kódoljuk, ami lehetővé teszi a szöveges adatok hatékony feldolgozását és elemzését
* Concatenate: Segítségével megoldhatjuk azt a problémát, hogy ha több oszlop adatait egyetlen oszlopba szeretnénk egyesíteni, például szöveges adatok esetén.

Az alábbi kódrészlet az adatok átalakítására mutat egy példát:

var dataProcessPipeline = mlContext.Transforms.Conversion.ConvertType(nameof(Tmdb.TmdbScore), nameof(Tmdb.TmdbScore), DataKind.Single)

.Append(mlContext.Transforms.Conversion.ConvertType("KeywordFloat", nameof(Tmdb.Keyword), DataKind.Single))

.Append(mlContext.Transforms.Conversion.ConvertType("GenreFloat", nameof(Tmdb.Genre), DataKind.Single))

.Append(mlContext.Transforms.Concatenate("Features", "KeywordFloat", "GenreFloat"))

.AppendCacheCheckpoint(mlContext);

Ebben az esetben a becsléshez 2 bemeneti adattag kerül felhasználásra, a Keyword és a Genre, ezek alapján történik meg a TmdbScore becslése. Mivel ezeknek az adatoknak a típusa int, ezért átalakításra van szükség: A DataKind.Single segítségével az ML.NET által a becslésekhez támogatott lebegőpontos számokká lehet őket alakítani, így jönnek létre a KeywordFloat és a GenreFloat mezők. Ugyan a TmdbScore értékének előrejelzésére irányul a becslés, viszont az adatok betanításához ennek értékeire is szükség van, viszont itt átalakítani nem szükséges, hiszen magának a bemeneti adatnak alapból float a típusa.

Az átalakított adatokkal létrejött új mezőket, a KeywordFloat-ot és a GenreFloat-ot végül egy mezőben, a Features-ben egyesítjük, ez fogja a bementi adatokat jelenteni a modell tanításához

* Algoritmus kiválasztása

Miután az adatokat megfelelő formába alakítottuk, kiválaszthatjuk a használni kívánt algoritmust attól függően, hogy mi a programunk célja. Az ML.NET függvénykönyvtárból több mint 30 algoritmus közül választhatunk.4 Ugyanazon probléma megoldására több algoritmus is alkalmas lehet, így ahhoz, hogy megtaláljuk az adott helyzetben a legjobban működőt, érdemes kipróbálnunk minél többet.

Az ML.NET által támogatott algoritmusok között megtalálhatóak a következő típusú metódusok:

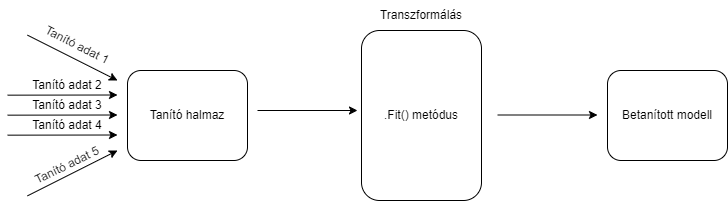
* Lineáris algoritmusok: A lineáris algoritmusok egy olyan modellt hoznak létre, amely a bemeneti adatok jellemzőit súlyozva és összegezve számítja ki a kimeneti értéket, ami ebben az esetben egy pontszám. A modell betanítása előtt normalizálni kell a jellemzőket ezen algoritmusok esetében, mivel így megakadályozható, hogy egyes jellemzők jobban befolyásolják a model működését mint a többi.[[10]](#footnote-10)
* Döntési fa algoritmusok: Általánosságban véve rendkívül pontos eredményeket lehet velük elérni, döntések sorozatát tartalmazzák a kimeneti érték megállapításához. Működésükhöz a többi algoritmushoz viszonyítva nagyobb erőforrást vesznek igénybe. A lineáris algoritmusokhoz viszonyítva nagy különbségük, hogy a jellemzőket nem kell normalizálni.6
* Mátrix-faktorizációk: A collaborative filtering rendszerekhez használatos algoritmusok.6 A collaborative filtering, vagy magyarul együttműködő szűrő rendszerek egy személynek különbőző más személyek, tárgyak, vagy információk iránti vonzalmát jelzik előre annak segítségével, hogy összevetik az adott személynek az érdeklődési körét az emberek többségének érdeklődési körével, így megkapják, hogy az adott személynek mi fog tetszeni az alapján, hogy más, hasonló ízlésű emberek mit kedveltek.[[11]](#footnote-11)
* Meta-algoritmusok: Ezen algoritmusok segítségével bináris osztályozókból többosztályos osztályozók készíthetőek.6
* K-közép: Klaszterezéshez használt algoritmusok.6 A gépi tanulás lehet ellenőrzött vagy ellenőrizetlen. Az ellenőrzött tanulás esetén előre meghatározott osztályaink/kategóriáink vannak, és minden egyes előforduló példát a megfelelő osztályba soroljuk be. Tehát maga a cél az, hogy a példákból általánosítani lehessen annak érdekében, hogy egy új példa melyik osztályba tartozzon. Ezt a feladatot osztályozásnak is nevezik. Az ellenőrizetlen tanulásnál van egy nagy különbség az osztályozáshoz képest, mivel itt gyakran az a cél, hogy magának az osztályozónak kell kitalálnia az osztályokat.[[12]](#footnote-12)
* Fő összetevő elemzése: Hibakeresésre, anomáliák felderítésére használatos algoritmusok.6
* Naiv Bayes: Többosztályos osztályozók esetében érdemes használni ezen algoritmusokat abban az esetben, mikor a jellemzők függetlenek egymástól, illetve kis mennyiségű adat áll rendelkezésünkre a model betanításához.6
* Prior trainer: A korábbi algoritmusokhoz képest ez egy egyszerűbb osztályozási problémára ad megoldást, hiszen csak bináris osztályozáshoz használhatóak. Sok esetben ilyen típusú algoritmusokat futtatnak először, mivel ez már ad egy viszonyítási értéket. Utána ha futtatunk más, különböző algoritmusokat, akkor azoktól ennél már jobb eredményt várunk el.6
* Vektorgépek támogatása: Ellenőrzött gépi tanulási algoritmusok, velük kapcsolatos megoldandó probléma, hogy hogyan lehet őket hatékonyan használni nagyobb adathalmazok esetében.6
* Legkisebb négyzetek: A lineáris regressziónak ez egyik leggyakrabban használt felhasználási módszere. Veszteségfüggvénnyel méri a modell hibáját, azaz azt, hogy mennyire tér el a modell által előrejelzett érték a ténylegesen megfigyelt értéktől.6
* Modell tanítása

A fentiek közül bármelyik típusú algoritmusból is választunk, ténylegesen csak akkor kerül végrehajtásra, ha meghívjuk a tanító adatok halmazára a .Fit() metódust. Ez az a pont ahol a modell tanítása, tehát maga a modellképzés történik.6

A modellképzés folyamatában a tanító adatok lesznek a bemeneti adatok, majd a modell tanítása, mint egy transzformátor jelenik meg. A transzformálás után kimenetként a képzett modellt kapjuk, amellyel előrejelzéseket, becsléseket készíthetünk a még ismeretlen adatokra vonatkozóan.6

ITransformer model = pipeline.Fit(trainingData);

(Az ML.NET dokumentációjából4 származó kódrészlet)



x. ábra: A modell betanításának a folyamata ML.NET-ben

* Modell kiértékelése

A modellünk kiértékelésénél először azt kell figyelembe vennünk, hogy milyen típusú algoritmust használtunk a modell betanításához. Más-más módszerekkel kell kiértékelnünk a bináris osztályozást megvalósító modellt, a többosztályos osztályozát megvalósító modellt, a regressziós modellt, a klaszterezést megvalósító modellt, illetve a rangsorolást, anomáliaészlelést és a mondatok hasonlóságának vizsgálatát megvalósító modelleket is.

A szakdolgozatomban a filmajánlórendszernél többosztályos osztályozási probléma, illetve regressziós osztályozási problémák jelentek meg, így ezek értékelésére térek ki részletesen.

A többosztályos osztályozó esetében az alábbi értékek használhatóak a kiértékelésre[[13]](#footnote-13):

* Mikro-pontosság: A mikro-pontosságot (vagy többosztályos pontosságot) úgy határozhatjuk meg, hogy a helyesen megjósolt példányokat viszonyítjuk a teljes adathalmazhoz, tehát ezzel a módszerrel kapunk egy olyan arányszámot, ami a helyesen megjósolt példányok hányadát jelenti.9 A mikro-pontosság minden esetet egyenlően kezel függetlenül az osztálytól, ezáltal a modell teljes teljesítményének vizsgálatában játszik nagyobb szerepet, hiszen egy egész, átfogó képet kaphatunk vele a modell pontosságáról.[[14]](#footnote-14) A kapott érték minél közelebb van 1-hez, annál jobb, hiszen annál pontosabb a vizsgált modell.9
* Makro-pontosság: A mikro-pontossággal ellentétben a makro-pontosság osztályszinten vizsgálva jelenti az átlagos pontosságot. Tehát az osztályok pontosságait egyenként kiszámítjuk, majd magát a makro-pontosságot ezeknek a kiszámolt pontosságoknak az átlagaként kapjuk meg. Ezáltal a makro-pontosság mérésénél minden osztály, az osztályok gyakoriságának figyelembevétele nélkül egyenlően járul hozzá a végleges pontosság meghatározásához, pl. 20 osztály esetében minden egyes osztály 1/20-addal befolyásolja a kapott értéket. A mikro-pontossághoz hasonlóan itt is a kapott érték minél közelebb van 1-hez, annál jobb, hiszen annál pontosabb az aktuálisan vizsgált modell.9
* Log-veszteség: Az osztályozási modell teljesítménye olyan formában mérhető vele, hogy a log-veszteség értéke annál jobban növekszik, minél jobban eltér a megjósolt eredmény a ténylegestől. Tehát minél kisebb a los-veszteség értéke, a modellünk annál pontosabb becslésre képes.9
* Log-veszteség csökkentése: A log-veszteség csökkentésénél a véletlenszerő találhatáshoz viszonyítunk. Tehát a modellünket azzal hasonlítjuk össze hogy az adott modell mennyivel működik hatékonyabban annál, mintha csak véletlenszerűen megpróbálnánk az értékeket eltalálni. A kapott szám mínusz végtelen és 1 közé kell hogy essen, és minél nagyobb ez a szám, annál hatékonyabb a modellünk. Ha pl. 0,1 a kapott érték, akkor 10%-kal működik jobban a modell a véletlenszerű találgatásnál.9

A Microsoft ML.NET keretrendszerhez kapcsolódó, a modell kiértékeléséről szóló dokumentációja9 szerint ha a felsorolt hatékonyságmérési módszerek közül csak egyet lehetne választani, akkor a legtöbb esetben mindenképpen a mikro-pontosságot a legérdemesebb.

Az alábbi kódrészlet a mikro-pontosság kiszámítását és kiíratását mutatja be:

var predictions = model.Transform(data);

var metrics = mlContext.MulticlassClassification.Evaluate(predictions);

Console.WriteLine($"MicroAccuracy: {metrics.MicroAccuracy}");

Regressziót használó osztályozók esetében az alábbi módszerek használhatóak a kiértékelésre:

* R-négyzet: Annak a mérőszáma, hogy a becsült értékek mennyire egyeznek meg a tesztadatokkal. A kapott érték mínusz végtelen és 1 között lehet. Ezt a tartományt 3 részre tudjuk osztani: ha 0 és 1 között van az érték, akkor a modell hatékonyabban működik a véletlen találgatásnál, minél nagyobb ez a szám, annál pontosabb. Ha pont 0 az értéke, akkor megegyezik a véletlenszerű találgatással. Mínusz érték esetén pedig rosszabbul működik a betanított modell, mintha egyszerűen véletlenszerűen találgatna.9
* Abszolút veszteség: Szintén arra vonatkozó mérőszám, hogy a becsült értékek mennyire állnak közel a valóshoz. Minden egyes értékre kiszámításra kerül a hiba mértéke, amely a becsült érték és a tényleges közötti abszolút távolság. Végül ezekből az értékekből számolunk egy átlagot, ez lesz a kapott abszolut veszteség. Minél közelebb van a szám a 0-hoz, annál pontosabb a modell.9
* Négyzetes eltérés: A regressziós egyenes és a tesztadatok értékkészletének kapcsolatát vizsgálja úgy, hogy a pontok és az egyenes közötti távolságokat négyzetre emeli. A négyzetre emelés miatt a nagyobb távolságok nagyobb súlyt kapnak. A kapott érték minden esetben 0 vagy pozitív, viszont minél közelebb van a 0-hoz, annál jobban működik a modell.9
* RMS-veszteség: A négyzetes eltérés négyzetgyöke, ezáltal könnyebben értelmezhetővé teszi a hibamérést. Minél közelebb van a 0-hoz, annál pontosabb a vizsgált modell.9
* A modell kimentése, használata

Az elkészített modellek kimentésére is lehetőséget kínál az ML.NET keretrendszer. Alapesetben a program minden futáskor új modellt generál az adott, aktuális jellemzők alapján, ez a memóriában kerül tárolásra addig a pontig, ameddig a program fut. Ha az elkészített modellünket a későbbiekben is szeretnénk használni, tesztelni, esetleg más programokban felhasználni, akkor kimenthetjük azt egy .zip kiterjesztésű tömörített fájlba.[[15]](#footnote-15)

A modell mentéséhez a Microsoft ML.NET dokumentációja11 szerint 2 dologra van szükség:

* Az elkészített modell ITransformerére, tehát gyakorlatilag a program azon részére, ahol a modellt betanítottuk a tanító adatokkal

var model = trainingPipeline.Fit(data);

* Az ITransformer várható bemenetének a DataViewSchema-ja, tehát a bemeneti adatok sémája

var data = mlContext.Data.LoadFromEnumerable(movies, schemaDef);

A modell kimentése a fenti változók segítségével:

mlContext.Model.Save(trainedModel, data.Schema, "model.zip");

(Az ML.NET dokumentációjából11 származó kódrészlet)

A kimentett modell visszatöltéséhez egy DataViewSchema és egy ITransformer típusú változóra van szükségünk11:

DataViewSchema modelSchema;

ITransformer trainedModel = mlContext.Model.Load("model.zip", out modelSchema);

(Az ML.NET dokumentációjából11 származó kódrészlet)

Az ITransformer típusú változóba betöltjük a korábban kimentett, modellt tartalmazó zip fájlt, és ezzel gyakorlatilag megkapjuk a modell sémáját is.

* Algoritmusok hatékonyságának összehasonlítása

A szakdolgozatom alapját egy folytonos érték becslése adja, melynek segítségével lehetséges értékelni a filmeket abból a szempontból, hogy mennyire tetszene az adott felhasználónak. Így az ML.NET-ben elérhető, különböző becslő algoritmusokat fogok összehasonlítani hatékonyság szempontjából.

A pontosság mérését a MovieRecommendationSystem MeasureAccuracy nevű osztályában valósítottam meg. A hatékonyság méréséhez az R^2 mérőszámot fogom használni, és a Tmdb pontszám becslését megvalósító DecTreeForTmdb osztály kódját alakítottam át az egyes becslési módszerek esetén olyan formába, hogy az adathalmaz minden elemére lefusson a becslés, és alkalmas legyen a pontosság mérésére.

Az R^2, mint a pontosság mérésére vonatkozó mérőszám már korábban említésre került, viszont az említett formában csak azt mutatta meg, hogy a betanítás után a modell mennyire pontosan tudja megbecsülni a betanító adatokat. Az én adatbázisom 100 elemű, viszont a becslésekhez ennek csak egy részét használtam fel, így létrehoztam egy külön, a modellek R^2 pontosságának mérésére szolgáló metódust annak a problémának a kiküszöbölésére, hogy ne csak azt lehessen megnézni, hogy a tanító adatokon milyen pontossággal működik, hanem azt is hogy az egész adatbázis adataira hogyan működik. A becslések a filmek Tmdb pontszámára vonatkoztak, ami minden film esetén ismert az adatbázisban, ezért könnyen össze lehetett őket vetni a becsült értékekkel.

A vizsgálat szempontja tehát a következő volt: A 100 darab film Tmdb pontszámának, tehát 1 és 10 közötti lebegőpontos értékek megbecslése egy 20 elemű tanítóhalmaz alapján.

A kapott eredmények R^2 pontosságai a következőek voltak:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Fast tree | Fast tree Tuned | Sdca | Sdca Tuned | Lbfgs Poisson R. | Lbfgs Poisson R. Tuned | Gam | Gam tuned | Online Gradient Descent | Online Gradient Descent tuned |
| 1. | -69,63 | 0,114 | 0,07 | 0,107 | -0,17 | -0,190 | -69,63 | -69,63 | -15,32 | -5,74 |
| 2. | -69,63 | 0,122 | 0,08 | 0,123 | -0,17 | -1,822 | -69,63 | -69,63 | -11,29 | -5,716 |
| 3. | -69,63 | 0,12 | 0,11 | 0,128 | -0,172 | -0,163 | -69,63 | -69,63 | -12,342 | -5,08 |
| 4. | -69,63 | 0,134 | 0,098 | 0,113 | -0,17 | -1,586 | -69,63 | -69,63 | -8,227 | -4,399 |
| 5. | -69,63 | 0,123 | 0,103 | 0,114 | -0,17 | -0,846 | -69,63 | -69,63 | -14,286 | -5,139 |

Minden algoritmust lefuttattam úgy, hogy a modellt különböző paraméterekkel megpróbáltam finomhangolni, illetve ezen paraméterek nélkül is. A finomhangolás során véletlen számokat generáltam bizonyos tartományokon belül a különböző paraméterekhez. A tartományoknál megpróbáltam figyelembe venni, hogy 20 elemszámú tanító halmazom van, tehát viszonylag kis méretű, és ehhez viszonyítva próbáltam az alsó és felső határokat meghatározni. Mindegyik algoritmusnak a finomhangolt verziójához 200 alkalommal generáltam véletlen számokat minden egyes parameter esetén, ezzel próbáltam megkeresni azokat a parameter értékeket, amelyekkel a leghatékonyabban működik a modell, és az ezekhez tartozó legmagasabb R^2 értékek kerültek a végén megjelenítésre, illetve a táblázatba.

Az R^2 mérőszám értéke mínusz végtelentől 1-ig terjedhet. Minél magasabb az érték, tehát minél közelebb van 1-hez, annál hatékonyabban működik a modell. Ha negatív értéket vesz fel, az azt jelöli, hogy a modell rosszabbul működik mint a véletlen találgatás. Ezen információk ismeretében az alábbi következtetések mondhatóak el a mérések alapján:

Finomhangolás nélkül egyetlen algoritmus tudott jobb eredményt hozni, mint a véletlen találgatás, ez pedig az Sdca volt. Az összes többi algoritmus még rosszabbul teljesített, mint a véletlen találgatás.

Finomhangolással az LbfgsPoissonRegression-ön és a Gam algoritmuson kívül mindegyiken lehetett javítani, és jobb értékeket lehetett elérni, mint finomhangolási paraméterek használata nélkül, viszont csak a Fasttree algoritmus tudott a véletlen találgatásnál jobb értékeket hozni.

Finomhangolás után a Fasttree és az Sdca algoritmusok közel azonos pontosságot hoztak, az 5 darab mérés eredményéből átlagot számítva az Sdca esetén 0,117-et, a Fasttree esetén 0,1226-ot kapunk, tehát a Fasttree algoritmus működött az összes közül a leghatékonyabban.

A fenti eredmények alapján tehát kijelenthető, hogy az említett problémára a Fasttree algoritmus adta a legpontosabb becsléseket az adott paramétertartományokban megtalált legjobb paraméterértékekkel. Viszont a 2 algoritmus eredményei rendkívül közel vannak egymáshoz, szóval a tartományok határával való további kísérletezéssel, vagy akár a tanítóhalmaz elemszámának növelésével/csökkentésével változni fog a pontosság, így akár az is, hogy melyik algoritmus teljesít jobban a másiknál.

Források:

<https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning>

<https://en.wikipedia.org/wiki/ML.NET>

<https://learn.microsoft.com/hu-hu/dotnet/machine-learning/how-does-mldotnet-work>

<https://dotnet.microsoft.com/en-us/learn/ml-dotnet/what-is-mldotnet>

<https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet>

<https://medium.com/@audaciatech/an-introduction-to-ml-net-and-the-functions-it-performs-8eeae0d94641>

<https://dl.acm.org/doi/abs/10.1145/358916.358995>

<https://www.sciencedirect.com/science/article/abs/pii/S0950705106000189>

<https://link.springer.com/article/10.1007/s10489-021-03041-7>

<https://vitalflux.com/micro-average-macro-average-scoring-metrics-multi-class-classification-python/#What_Why_of_Micro_Macro-averaging_and_Weighting_metrics>

A movies.db adatbázis Movies táblájában filmek, és a hozzájuk tartozó különböző adatok szerepelnek. 1 sor 1 filmet tartalmaz, illetve annak adatait. Viszont a tulajdonságok között több olyan mező is van, amely vesszővel elválasztott felsorolásokat tartalmaz.

Ezeknek a mezőknek a tartalmát ebben az állapotban csak egyben lehet felhasználni, ez pedig nem előnyös. Ha a filmeket össze szeretnénk hasonlítani, esetleg csoportosítani, akkor ez esetben arra van szükség, hogy a közös tulajdonságaikat vizsgáljuk meg: vannak-e egyáltalán, ha igen, akkor melyek azok? Amennyiben egy olyan tulajdonságot vizsgálunk, amelyhez tartozó mező több adatot is tartalmazhat (pl. egy film kulcsszavainak vizsgálata esetén a film nem csak egy kulcsszóval rendelkezhet, hanem felsorolás szerűen többel is), akkor arra rendkívül kicsi esély van, hogy az adott tulajdonságot vizsgálva a 2 film 100%-osan megegyezzen. Viszont ettől még lehet számos egyezés, azonban ha az adatokat egyben vizsgáljuk akkor ezeket nem tudjuk megtalálni, ezért szükséges ezen mezők lebontása, hogy a tartalmukban szereplő különböző adatokat külön-külön tudjuk kezelni, ne pedig egyben.

Tehát a mezőket le kell bontani, hogy kezelhetőek legyenek. Ebben az esetben szükség van egy külön táblára, amiben az éppen lebontott mező összes adata szerepel ismétlődések nélkül úgy, hogy minden egyes adat egy új sor. Tehát ahol eddig vesszők voltak, azok az adatok külön-külön új sorba kerülnek. Ahhoz, hogy az így kapott adatokat hozzá tudjuk rendelni a filmekhez, mivel több-több kapcsolat van az adatok között, így szükség lesz annyi kapcsolótáblára is, ahány új táblánk keletkezett az adatok lebontása során. A TableInserts nevű C# program olyan egyszerű konzolos alkalmazás, amely ezen táblák, illetve kapcsolótáblák feltöltését végzi el.

**A lebontás folyamatának általános lépései**

1. Az adott lebontandó mező összes adatának kigyűjtése egy listába
2. A listából az ismétlések kiszűrése
3. A lista maradék elemeihez egyedi id-t rendelni, így megkapjuk az összes lehetséges adatot ami hozzá lehet rendelve a filmekhez
4. Az így kapott adatok feltöltése az adatbázis megfelelő táblájába
5. A filmek még le nem bontott tulajdonságainak összevetése a már kiszűrtekkel
6. Ha az összevetés során egyezést találunk, akkor a film id-jához hozzárendeljük annak az elemnek az id-ját, amelynél az egyezést találtuk, ezek lesznek a kapcsolótábla adatai
7. A kapott adatokkal a kapcsolótábla feltöltése

A lebontandó mezők a Keywords, Languages, Directors, Countries és Genres. A program az adatbázis movie tábláját használja bemeneti adatként, és az előbb említett táblákat, illetve az azokhoz kapcsolódó kapcsolótáblákat tölti fel. Mivel egy SQLite adatbázisból olvas, és abba is ír, ezért kapcsolatot kell teremteni a program és az adatbázis között, ez az SqlConnector nevű osztályban valósul meg.

**SqlConnector osztály**

Az SQL kapcsolat létrehozása mellett ebbe az osztályba kerültek az SQL kapcsolatot igénylő lekérdezéseket, táblamódosításokat tartalmazó metódusok is. Az osztály tartalmaz egy collectMovie nevű metódust, amely a filmek szükséges adatait kérdezi le és tárolja el a Program.cs-ben definiált Movie típusú movies listában, tehát itt kerülnek felhasználásra a Movies osztályban megadott változók.

A PrepareKeywords, PrepareLanguages, PrepareDirectors, PrepareCountries, és PrepareGenres metódusok a film adott jellemzőit kigyűjtik egy listába, ahol minden egyes elem, ami az adatbázis movie táblájában vesszővel volt elválasztva, az ide új listaelemként kerül be. Ezek a metódusok szolgálnak a Keywords, Languages, Directors, Countries és Genres táblák alapjául. Az összes filmre vonatkozó adat egy listába kerül bele, hiszen a lényeg, hogy egy helyen legyen az összes lehetséges adat, ami majd később hozzárendelésre kerül a filmekhez. A listában ezen a ponton még sok ismétlődés lesz, viszont ezek a későbbiekben kiszűrésre kerülnek, és a lista minden eleméhez egyedi azonosítót, id-t rendelünk. A Program.cs osztályban létrehozott data listában kerülnek majd eltárolásra a metódusokban kigyűjtött adatok, a különböző prepare metódusok a Table osztályban kerülnek meghívásra. Erre a célra elég egyetlen lista is, hiszen egyszerre mindig csak egy táblát töltünk fel, így csak az ahhoz szükséges adatokat kell eltárolnunk.

Az osztály tartalmaz még egy RemoveWhitespace metódust is. Erre azért van szükség, mivel az előbbiekben említett metódusoknál a program a vesszőket figyeli mint határolópont, viszont az adatbázis eredeti mezőiben ahonnan dolgozik, ott vesszővel és szóközzel vannak határolva az adatok, így a data listába kigyűjtve a legtöbb listaelem egy szóközzel fog kezdődni, hiszen így került kigyűjtésre. Ez a metódus végigmegy az adott lista minden elemén amire meghívjuk és amennyiben az adott elem szóközzel kezdődik, akkor törli az első karakterét, tehát a szóközt a szöveg elejéről.

Végül az osztály egy ExecuteInserts nevű metódussal zárul, ami bemeneti paraméterként egy listát kap meg, amelynek minden egyes eleme egy-egy táblafrissítő parancs, amelyet a Program.cs osztályban állítunk össze. A metódus végigmegy a lista elemein és sorra végrehajtja az SQL parancsokat. Hiba esetén kiírja, hogy „An error occurred during table upload”.

**Program.cs osztály**

A Main metódusban felépítésre kerül az adatbázis eléréséhez szükséges út, az SqlConnector, FillUps, Algorithms, és Table osztályok példányosítása, illetve a program működéséhez szükséges változók:

* Egy int típusú id lista, amelyben a Keywords, Languages, Directors, Countries és Genres táblák feltöltéséhez használt id-k kerülnek majd tárolásra
* A korábban már említett string típusú data lista
* További két int típusú lista, a finalMovieID és a finalDataID, melyekre majd a kapcsolótáblák feltöltésére szolgáló parancsoknál lesz szükség,
* A string típusú cmds, melybe a végrehajtandó SQL utasítások kerülnek bele, ezeket fogja végrehajtani a korábban említett ExecuteInserts metódus

A definiált változók után meghívásra kerül a már bemutatott CollectMovie metódus, tehát a movies listában eltároljuk a filmek lebontandó adatait mindenféle változás nélkül.

Ezután meghívásra kerül a Table osztály Options metódusa.

**Table.cs osztály**

A Table osztály az Algorithms, SqlConnector és FillUps osztályok metódusaival dolgozikj.

A Table osztály tartalmazza az Options metódust. A felhasználó itt tudja kiválasztani, hogy a Keywords, Movie\_Keywords, Languages, Movie\_Languages, Directors, Movies\_Directors, Countries, Movies\_Countries, Genres és Movies\_Genres táblák közül melyiket szeretné feltölteni. Az opciók 0-9-ig vannak megszámozva, a felhasználó a megfelelő szám beírásával tud választani közülük. A metódus további része egy switch case szerkezet. A Table.cs osztály minden feltölthető táblához tartalmaz egy a tábla nevével megegyező nevű metódust, amelyben az adott tábla feltöltéséhez szükséges metódushívások vannak elrendezve. A switch case szerkezettel a kiválasztott szám alapján ezek közül a metódusok közül hívunk meg egyet.

Ezeknek a metódusoknak két típusa van:

* Táblát tölt fel: Genre, Keyword, Language, Director, Country
* Kapcsolótáblát tölt fel: Movies\_Genres, Movie\_Keywords, Movie\_Languages, Movies\_Directors, Movies\_Countries

Táblát tölt fel

Amelyek táblát töltenek fel, azok 6 darab metódushívást tartalmaznak. Az alábbiakban a keywords metódus tartalma kerül bemutatásra, de a Genre, Language, Director, country metódusok is ugyanígy néznek ki, természetesen ahol szükséges ott az adott táblához igazítva:

1. Az adott táblához kapcsolódó, már korábban részletezett prepare metódus meghívása.

A Keywords esetén a PrepareKeywords. A metódushíváskor a data listát adjuk át neki referencia szerint, hogy a metódusban a listán végzett módosítások a metódus futásának befejeztével ne vesszenek el. A futása végén megkapjuk a data listában az összes kulcsszót, ömlesztve.

1. Az Algorithms osztály Matches metódusának meghívása.

A metódusnak átadjuk az előzőleg feltöltött data listát, illetve létrehozunk benne egy átmeneti string típusú listát (list\_ok). Végigmegyünk a data minden elemén, és ami még nem szerepel a list\_ok-ban, azt hozzáadjuk, ha pedig már szerepel benne az adott elem akkor megyünk a következőre. A metódus visszatérési értéke egy string típusú lista, így a list\_ok-ot fogjuk belőle visszaadni. A metódushívás helyén a data listát egyenlővé tesszük magával a metódushívással, tehát gyakorlatilag a list\_ok lista tartalmával.

1. Az Algorithms osztály IdGenerate metódusának hívása.

A metódus arra szolgál, hogy az előző lépésben már ismétlődésmentessé szűrt data lista elemeit egyedi azonosítóval, kulccsal lássa el. Paraméterként megkapja a data listát, illetve referencia szerint a main függvényben már létrehozott int típusú id listát. A kulcs generálás egy for ciklus segítségével történik, 1-től indul és egyesével növekszik.

1. Az adott táblához kapcsolódó FillUp metódus meghívása a FillUps osztályból.

A keywords esetén FillUps.Keywords. Ezzel a metódussal történik az adatbázis adott táblájának feltöltéséhez szükséges utasítások összeállítása, jelen esetben a Keywords feltöltéséhez szükséges SQL táblafeltöltő utasítások összeállítása LINQ használatával. A tábla feltöltéséhez felhasználjuk az előzőleg összekészített data és id listát, ezeket megkapja a metódus paraméterként, továbbá referencia szerint átadjuk neki a korábban már említett string típusú cmds listát, mivel ebben kerülnek majd tárolásra a parancsok.

1. A korábban már említett SqlConnector osztályban található ExecuteInserts metódus meghívása.

Az előző lépésben összekészített cmds listát átadjuk a metódusnak, és egy for ciklus segítségével mindegyik összekészített utasítás végrehajtásra kerül.

1. Végül a cmds listát a cmds.Clear() használatával kiürítjük.

Azért szükséges, mert ha egy futtatáson belül szeretnénk több táblát is feltölteni, akkor a korábban már végrehajtott utasítások a következő tábla feltöltésénél újra megpróbálnának végrehajtódni, viszont hibát kapnánk, hiszen ezek az adatok léteznek már a táblában.

Kapcsolótáblát tölt fel

Azok a metódusok, amelyek kapcsolótáblát töltenek fel, azok tartalmazzák azokat a metódushívásokat is, amelyek az előbbiekben, a sima tábláknál kerültek bemutatásra. Az alábbiakban a Keyword\_Movie metódus tartalma kerül részletezésre, de a Genre\_Movie, Language\_Movie, Director\_Movie, és Country\_Movie metódusok is ugyanígy néznek ki, természetesen ahol szükséges ott az adott táblához igazítva:

1. mode változó beállítása.

A mode változó segítségével fogjuk tudni később az Algorithms osztály createData metódusánál eldönteni, hogy a filmek melyik adatával dolgozunk, így ez az érték mindegyik kapcsolótábla esetén más.

1. PrepareKeywords meghívása
2. Az Algorithms osztály Matches metódus hívása
3. Az Algorithms osztály IdGenerate meghívása
4. Az Algorithms osztály CreateData függvény meghívása.

A metódus megkapja paraméterként az id, data, movies listát és a mode változót, továbbá referencia szerint átadjuk neki a finalMovieID és a finalDataID int típusú listákat. Ezekbe kerülnek majd bele azok az adatok, amelyekkel a kapcsolótáblát feltöltjük, tehát ezeket adjuk majd át az adott táblához tartozó FillUp metódushoz, mint az a sima tábláknál is történt.

A függvényben létrehozásra kerül egy string típusú temp lista, ezután az Algorithms osztály copy nevű metódusát hívjuk meg, átadva neki a movies listát, referencia szerint pedig a temp listát, illetve a mode változó értékét. A copy függvény egy switch case szerkezetet tartalmaz, a mode változó értéke alapján dönti el, hogy a temp listát a movies lista melyik adataival töltse fel. (Például a keyword\_movie kapcsolótábla feltöltésekor a program futásának ezen a pontján értelemszerűen a movies listából a keyword adatokra van szükség. A Keyword\_Movie függvény elején a mode változót 0-ra állítottuk be, ezért a copy metódus ha a mode értéke Mode.Keyword enum érték, akkor a temp listát a movies lista keyword adataival tölti fel, ami az adatbázisunk movie táblájának keyword mezőinek értékét tartalmazza, minden egyes filmhez a vesszővel elválasztott értékeket amiket eredetileg elkezdtünk lebontani).

A copy metódusból a CreateData-ba visszatérve végigmegyünk a temp listán. Két egymásba ágyazott for ciklus segítségével minden egyes elemét összehasonlítjuk a data lista összes elemével (a data lista a kigyűjtött, ismétlődésmentes kulcsszavakat tartalmazza, illetve az id lista az ezekhez rendelt egyedi kulcsot). Amennyiben egyezés van, akkor a finalMovieID listához hozzáadjuk az aktuálisan vizsgált film azonosítóját, (movies[index].Id), a finalDataID listához pedig annak a kulcsszónak az azonosítóját az id listából, amellyel az egyezés történt. Így létrejönnek a kapcsolótábla feltöltéséhez szükséges adatok.

1. FillUp\_MoviesKeywords meghívása
2. ExecuteInserts metódussal a parancsok végrehajtása
3. cmds.Clear() használatával a lista kiürítése

A szakdolgozatom fő programja a MovieRecommendationSystem program, amely egy grafikus felületű C# alkalmazás. A program célja, hogy a felhasználó számára egy olyan filmet ajánljon, ami az ő érdeklődési körébe esik, az ő igényeinek felel meg. A felhasználó érdeklődési körének felmérése céljából a program indításkor kérdéseket tesz fel a felhasználónak, amelyek megválaszolása után tudja meghatározni az alkalmazás az ajánlott filmet. A program szintén a movies.db adatbázisból dolgozik, így ezt használja az ajánlott film kiválasztásához is.

Movie osztály

Mivel a filmek pontos osztályozásához, majd ajánlásához minél több információra van szükség, így az összes rendelkezésünkre álló adatra szükség van a filmek esetén. Így már nem csak a TableInserts által lebontott mezőkhöz kapcsolódó adattagokat kell tartalmaznia a Movie osztálynak, hanem a film összes tulajdonságához szükség van egy adattagra, amelyben eltároljuk az adott információt. A lebontott adatokat táróló adattagok listák lesznek, hiszen ebben az esetben több ugyanolyan típusú információt is el kell tárolnunk egy adott film esetében, a többi adattag viszont sima változó lesz.

A lebontott adattagokhoz kapcsolódóan adattagonként 3 lista került létrehozásra, melyek pl. a Genre adattag esetén a következőek lesznek:

* Genre: A kapcsolótábla adatait tárolják, tehát ez esetben a műfajok ID-jait, tehát int típusú adattag
* GenreString: A Genre adattag és a Genre tábla segítségével az ID-k segítaségével beazonosítjuk a műfajok neveit
* GenreStringWithCommas: A műfajok neveinek vesszővel elválasztott felsorolása táblázatos megjelenítéshez

A MovieRecommendationSystem és a TableInserts program összekapcsolása

A TableInserts programnak az a funkciója, hogy az adatbázis eredeti Movies táblájában szereplő olyan tulajdonságokat, amelyek több adatot is tartalmaznak vesszővel elválasztva, azokat lebontsa, és segédtáblákat, kapcsolótáblákat készítsen hozzájuk, amelyeknek a tartalmait hozzárendeljük a filmekhez. Tehát a program gyakorlatilag egy segédprogrogramként funkcionál ahhoz, hogy ezen adatokat felhasználva a MovieRecommendationSystem működhessen.

A két program összekapcsolása érdekében a TableInserts program metódusait egy TableInsertsLibrary nevű függvénykönyvtárba rendeztem ki ahhoz, hogy a már meglévő metódusok felhasználhatóak legyenek a fő programban is. A programot néhány helyen módosítani, vagy bővíteni kellett annak érdekében, hogy jól tudjon illeszkedni a fő programhoz. Ezek a módosítások a következők voltak:

A FillUps osztályban létrehozásra került egy UpdateDB metódus, amelynek parancsai törlik az összes segédtáblát és kapcsolótáblát amit a program létrehozott, tehát a Movies táblán kívül gyakorlatilag minden táblát, majd újra létrehozza ezeket. Szóval a táblák újrafeltöltését teszi lehetővé az algoritmus, amelyre a program működése során akkor lesz szükség, ha filmeket adunk hozzá, törlünk, vagy módosítunk az adatbázisunkban, vagy teljes adatbázist cserélünk tallózás segítségével, mivel ezekben az esetekben a filmekhez hozzá kell igazítanunk a többi tábla tartalmát is.

Szintén a FillUps osztályban a táblafeltöltő parancsokat összekészítő metódusokban az INSERT parancsok BEGIN és COMMIT utasítások közé kerültek, ezáltal egy tranzakción belül hajtódnak végre az azonos táblát feltöltő parancsok, ezzel növelve a táblák feltöltésének sebességét.

A Table osztályhoz hozzáadásra került egy InsertAll nevű metódus, a fő programban ezt hívjuk meg az Options helyett, mivel az Options metódus egyszerre csak egy táblát tudott feltölteni, és azt is külön ki kellett választani hogy melyik legyen az. Ez a TableInserts program teszteléséhez jól tudott jönni, de a fő program esetében minden táblát fel kell töltenünk egyszerre. Ezt a feladatot oldja meg az InsertAll nevű metódus.

PropertiesForDecTree osztály

A programban a becslésekhez az ML.NET függvénykönyvtárat fogom használni. A függvénykönyvtárnak viszont hiányossága, hogy a modell betanításához listák sajnos nem használhatóak, így abban az esetben, ha a filmeknek egy olyan, több adatból álló tulajdonságát szeretnénk használni a betanításhoz, amelyet korábban a TableInserts programmal lebontottunk, akkor egyenlő elemszámú tömbökre van szükség. A PropertiesForDecTree osztály adattagjai ezeknek a tömböknek a létrehozásához, feltöltéséhez lesznek szükségesek.

internal class PropertiesForDecTree

{

public List<string> GenreAll { get; set; }

public List<string> KeywordAll { get; set; }

public List<string> LanguageAll { get; set; }

public List<string> DirectorAll { get; set; }

public List<string> CountryAll { get; set; }

public int[][] GenreContains { get; set; }

public int[][] KeywordContains { get; set; }

public int[][] LanguageContains { get; set; }

public int[][] DirectorContains { get; set; }

public int[][] CountryContains { get; set; }

}

A listákban kerül eltárolásra az egyes tulajdonságok esetén az összes lehetséges opció, tehát például a GenreAll lista az összes előforduló műfajt tartalmazza. A listák az SqlConnectorban található Get metódusban kerülnek majd feltöltésre. A kétdimenziós tömböknél pedig minden egyes filmet megvizsgálunk az adott jellemző minden egyes lehetséges értékére, ezek az értékek lesznek majd a tömbökben eltárolva.

SqlConnector osztály

Mivel a program adatbázisból dolgozik, így ugyancsak meg kell teremteni a kapcsolatot az adatbázissal, ami az SqlConnector osztályban valósul meg. Emellett itt történik az adatbázis movie táblájából az egyszerű, felsorolást nem tartalmazó mezők adatainak betöltése.

Az egyszerű adatok mellett viszont szükséges a TableInserts programmal már előkészített, lebontott formába került adatok beolvasása és eltárolása is, az osztály minden ilyen adattaghoz tartalmaz egy Fillup metódust, mint pl. a FillupGenre.

/// <summary>

/// A korábbiakban feltöltött Id adattag és a Movies\_Genres kapcsolótábla segítségével a Genre adattag feltöltésre kerül, itt még a műfajokat azonosító integerekkel

/// </summary>

/// <param name="movies">A movies listában tároljuk el a filmek műfajait</param>

public void FillupGenre(ref List<Movie> movies)

{

int index;

int data;

SQLiteDataReader reader = null;

SQLiteCommand command = connection.CreateCommand();

command.CommandText = "SELECT \* FROM Movies\_Genres";

reader = command.ExecuteReader();

while (reader.Read())

{

index = Convert.ToInt32(reader["Movie\_ID"]);

data = Convert.ToInt32(reader["Genres\_ID"]);

for (int i = 0; i < movies.Count; i++)

{

if (movies[i].Id == index)

{

movies[i].Genre.Add(data);

}

}

}

reader.Close();

}

A kapcsolótáblában lévő, jelen esetben a műfajokat azonosító integerek megfelelőek arra, hogy hozzárendeljük a filmekhez a különböző adatokat, műfajokat, viszont a megjelenítésükkel nem derül ki, hogy konkrétan melyik műfajról van szó. Tehát szükség van egy metódusra, amely minden filmhez hozzárendeli az azonosítókon kívül a tényleges adatot is, aktuálisan a műfajok nevét. Az osztály minden ilyen adattaghoz tartalmaz egy ilyen metódust, mint jelen esetben a műfajokhoz a GetGenre. Ez még nem a tényleges hozzárendelést végzi el, csak az ahhoz való előkészítést. Tehát a metódusban az egyes jellemzők id-jait és neveit kérdezzük le az adatbázisnak az aktuális jellemzőjéhez tartozó táblából, jelen esetben a Genre esetében a Genre táblából, és ezen adatokat paraméterként átadva meghívjuk az Algorithms osztrály megfelelő metódusát, jelen esetben a GenreToString-et, amely ténylegesen elvégzi az adatok neveinek a hozzárendelését a filmekhez.

Mivel a metódusban az egyes jellemzők esetén kigyűjtjük az összes lehetséges értéket, ezért ezeket eltároljuk a PropertiesForDecTree megfelelő adattagjaiban, ezeket fogjuk később felhasználni a tömbök feltöltéséhez.

/// <summary>

/// A korábbiakban megkapott integerek felhasználásával hozzárendeljük a műfajok nevét is a filmekhez

/// </summary>

/// <param name="movies">A movies lista megfelelő adattagjában fogjuk eltárolni a műfajok tényleges neveit is</param>

/// <param name="tableID">A műfajok integer azonosítót tároljuk benne</param>

/// <param name="tableData">A műfajok neveit tároljuk benne</param>

public void GetGenre(ref List<Movie> movies, List<int> tableID, List<string> tableData, ref PropertiesForDecTree prop)

{

SQLiteDataReader reader = null;

SQLiteCommand command = connection.CreateCommand();

command.CommandText = "SELECT \* FROM Genres";

reader = command.ExecuteReader();

while (reader.Read())

{

tableID.Add(Convert.ToInt32(reader["ID"]));

tableData.Add(reader["Genre\_Name"].ToString());

prop.GenreAll.Add(reader["Genre\_Name"].ToString());

}

reader.Close();

Algorithms alg = new Algorithms();

alg.GenreToString(ref movies,tableID,tableData);

}

Végül egy egyszerű SQL parancsot végrehajtó metódust is tartalmaz az osztály, ez az UpdateCommand nevet kapta, és az adatbázisnak a programon belül történő kezeléséhez fogjuk felhasználni a későbbiekben.

Algorithms osztály

A Movie típusú movies lista azon adattagjait, amelyek nem sima változók, hanem listák, inicializálni kell, ez az Algorithms osztályban történik az InitLists metódussal.

A PropertiesForDecTree osztály adattagjait, amiket majd később a becslésekhez használunk szintén az Algorithms osztályban inicializáljuk, illetve a tömböket itt is töltjük fel. Inicilizálásuk a listák esetében az InitListsForFillContains metódussal, a tömbök esetében az InitArraysForFillContains metódussal történik.

A feltöltés a FillContains nevű metódussal történik. A metódus minden egyes olyan jellemzőhöz tartalmaz programrészt, amelyet korábban a TableInserts programmal bontottunk le. Tehát például a műfajok esetében is, ahogyan már korábban említve volt, egyenlő elemszámú tömbök lesznek szükségesek a becslési modell betanításához.

A tömbök inicializálása a fent említett metódussal már megtörtént. A feltöltéshez pedig pl. a műfajok esetén minden egyes filmnél megnézzük minden egyes műfajnál, hogy az adott film olyan műfajú-e, és ha igen akkor az adott film esetén az adott műfajhoz 1-et rendelünk, különben pedig 0 értéket, így egyenlő elemű kétdimenziós tömböket kapunk. Ezeket az értékeket fogja tartalmazni a műfajok esetén a GenreContains tömb.

A feltöltéseket ugyanezzel a logikával az összes többi lebontott tulajdonságra elvégezzük a metódus további részében.

/// <summary>

/// A tömbök feltöltése minden egyes filmnél, azon belül minden egyes előforduló adat esetén 0 vagy 1 értékkel

/// </summary>

/// <param name="prop">A PropertiesForDecTree osztály példányosításának adattagjait töltjük fel</param>

/// <param name="movies">A filmeken megyünk végig, illetve azok adatait vizsgálva döntünk a 0 vagy 1 értékről</param>

public void FillContains(ref PropertiesForDecTree prop, List<Movie> movies)

{

for (int i = 0; i < movies.Count; i++)

{

for (int j = 0; j < prop.GenreAll.Count; j++)

{

for (int k = 0; k < movies[i].GenreString.Count; k++)

{

if (movies[i].GenreString[k].Contains(prop.GenreAll[j]))

{

prop.GenreContains[i][j] = 1;

break;

}

else

{

prop.GenreContains[i][j] = 0;

}

}

}

}

.

.

.

A korábban az SqlConnector osztályban már említett metódusok is az Algorithms osztályban találhatóak, amelyek a filmekhez a különböző jellemzők id-jai segítségévwel elvégzik a jellemzők neveinek is a hozzárendelését.

A filmeken végigmegyünk, majd az egyes filmek minden egyes műfajának adatait a Genres adattagból (amik egyelőre id-k) összehasonlítjuk a tableID lista minden elemével. Ha egyezés van akkor mivel a tableID és tableData listákban párhuzamos indexeléssel vannak letárolva az adatok, ezért az az adott filmhez tartozó GenreString adattaghoz hozzáadjuk a tableData lista adott indexű elemét.

/// <summary>

/// A GenreString adattag tényleges feltöltése az egyes filmek esetén

/// </summary>

/// <param name="movies">A movie lista GenreString adattagjába kerülnek az adatok</param>

/// <param name="tableID">A Genres táblából származó műfaj azonosítók (id-k)</param>

/// <param name="tableData">A Genres táblából származó műfajok tényleges nevei</param>

public void GenreToString(ref List<Movie> movies, List<int> tableID, List<string> tableData)

{

for (int i = 0; i < movies.Count; i++)

{

for (int j = 0; j < movies[i].Genre.Count; j++)

{

for (int k = 0; k < tableID.Count; k++)

{

if (movies[i].Genre[j] == tableID[k])

{

movies[i].GenreString.Add(tableData[k]);

}

}

}

}

tableID.Clear();

tableData.Clear();

}

Végül a LoadTableInserts metódus is ebbe az osztályba tartozik. Ez a metódus arra szolgál, hogy a TableInserts program dokumentációjánál már ismertetett metódusokat meghívja, ezáltal a TableInsertsnek a függvénykönyvtárba szervezett programrészei is lefussanak. Tehát tartalma lényegében szinte megegyezik a TableInserts Program osztályáéval a korábban részletezett módosításoktól eltekintve.

Az Algorithms osztály tartalmaz még 2 további metódust, az IsBlockbuster és az IsPopular metódust, amely a movies lista filmjeinek a metódusok neveivel megegyező nevű, bool típusú adattagjait töltik fel igaz vagy hamis értékekkel.

Az IsPopular metódusban a filmek népszerűségéből számolunk egy átlagot, majd végigmegyünk a lista összes filmjén, és ha az átlag felett van az adott film népszerűsége, akkor igaz értéket kap az aktuális adattag, ha pedig egyenlő az átlaggal, vagy alatta van, akkor hamis értéket.

Az IsBlockbuster metódus szintén végigmegy az összes filmen, és a filmek Budget és Revenue adattagjait használja fel arra, hogy eldöntse, hogy kasszasiker-e az aktuális film, vagy sem. Ha a film bevételéből (Revenue) kivonjuk a film költségvetését (Budget) és 0-tól nagyobb értéket kapunk, akkor az adott filmet kasszasikernek tekintjük. Ha kasszasiker, akkor igaz értéket kap az aktuális adattag, ha nem, akkor pedig értelemszerűen hamisat.

HandmadeLanguageDecTree osztály

A program tartalmaz egy programrészt annak a kézzel készített, film nyelvére vonatkozó döntési fának a megjelenítésére is, amelynek a hatékonyságát vizsgáltuk a géppel készített becsléshez viszonyítva. Ennek megvalósítására ebben az osztályban kerül sor, a kirajzolás a GraphViz szoftver segítségével történik. A GraphViz .dot formátumba formázott fájlból képes képet készíteni, így az osztály jelentős részében ennek a .dot fájlnak az elkészítése történik a fa feldolgozása közben.

A MainJson metódusban a fát először beolvassuk a json forrásfájlból, majd egy JObject-be alakítjuk az egész tartalmát. Ezután egy dot nevű string listába fogjuk összekészíteni a megjelenítéshez szükséges adatokat, amelyeket a fa feldolgozásával kapunk. Kezdetben elnevezzük a fát „DecisionTree”-nek, és a jsonObj JObject típusú változó segítségével kiolvassuk a fa első csomópontját, amely a root, majd hozzáadjuk a dot listához „Tmdb-score” címkével, mivel ez az első csomópont neve. Ezután meghívjuk a rekurzív ParseNode metódust, amely 3 paraméterrel rendelkezik: a dot lista, amelybe a feldolgozott információkat gyűjtjük, egy node nevű JToken változó, amely a fa aktuálisan vizsgált részét tartalmazza, és egy string típusú parent változó, amelybe az aktuális csomópont szülőjének a neve kerül eltárolásra.

Kezdetben az egész fára meghívjuk a metódust, szülőnek pedig a „root”-ot adjuk. Ha a node változó típusa Object (első futáskor az lesz), akkor a fának az aktuálisan vizsgált csomóponttól történő közvetlen elégazásai, illetve azoknak a további alsó (közvetett) elágazásai bekerülnek a JArray típusú conditionArray tömbbe. Tehát a metódus első futásakor az aktuálisan vizsgált csomópont az a legelső, azaz a Tmdb-score lesz, így a conditionArray elemei a fának ettől a csomóponttól lefelé lévő részei lesznek. Mivel a Tmdb-score elágazásnak 3 közvetlen elágazása van, így ebben az esetben a conditionArray tömb 3 elemű lesz. Egy foreach ciklussal ezeket az elágazásokat egyenként is megvizsgáljuk. A condition string változóban eltároljuk az aktuálisan vizsgált elágazás nevét, az outcome változóban pedig az aktuálisan vizsgált csomópont kimenetét, amely lehet 1 db kimenet (levél), több db kimenet (levél), vagy akár egy újabb csomópont is, amelyből további elágazások vezetnek. A metódus ezen a ponton 3 ágon haladhat tovább, attól függően, hogy milyen típusú a fent említett 3 közül a kapott kimenet. A Tmdb-score első elágazásának kimenete egy további csomópont (Budget), ezért generálunk egy új azonosítót ennek a csomópontnak, majd a dot listához hozzáadjuk, hogy a szülőből ebbe a csomópontba megyünk, az adott feltétel (elágazás, amely most a <=6.0) esetén. Továbbá az újonnan hozzáadott csomóponthoz címkeként hozzáadjuk a csomópont nevét, jelen esetben a „Budget”-ot. Ezután újra meghívjuk a ParseNode metódust, mivel a Budget csomópont elágazásait hasonlóan fel kell bontani, mint azt a Tmdb-score-ral tettük. A metódus paramétereit nézve a dot lista ebben az esetben sem változik, a fa lebontandó része, a JToken az előző csomópont lebontásakor kapott kimenet lesz (tehát most a Budget csomópont és annak elágazásai), a parentName pedig az új csomópontnak (Budget) generált azonosító.

Ha a csomópont kimenete nem egy másik csomópont, hanem egy vagy több levél, akkor is ugyanazzal a logikával kerülnek bele a dot fájlba az információk, mint ami a csomópont esetében be lett mutatva, csak ebben az esetben a leveleknek generálunk azonosítót, illetve nem hívjuk meg a ParseNode metódust, mivel ezek már tényleges kimenetek, így nincs mit tovább bontani. Továbbá ovális helyett téglalap alakú objektumként jelenítjük meg őket, jelezve, hogy ők kimenetek.

Ezzel a logikával a ParseNode metódussal végigmegyünk az egész json fájl tartalmán, majd a MainJson metódus végén a megjelenítéshez szükséges információkkal feltöltött dot listából készítünk egy dot fájlt, amiből GraphViz segítségével előállítjuk a megjeleníteni kívánt png képet.

Az osztály tartalmaz még egy CreateJsonPath metódust, ez a forrásfájl helyét határozza meg, amely szükséges a beolvasáshoz.

Main osztály

A program fő osztálya a Main, ami a program fő ablakát, főmenüjét jeleníti meg.

Az osztály fő metódusa a Main\_Load, ami akkor fut le, amikor az ablak megjelenítésre kerül, tehát mivel ez a fő ablak, így gyakorlatilag a program indulásakor. Ebben a metódusban történik a kapcsolódás az adatbázissal, a korábban bemutatott metódusok meghívása, az adatok kiolvasása az adatbázisból, illetve a korábban lebontott mezők adatainak tényleges hozzárendelése a filmekhez.

A Main osztály egy további metódusa, a CreatePath építi fel az adatbázis eléréséhez szükséges utat, ezzel elkerülhető az abszolut útvonal használata.

A főmenü tartalmaz egy Table, egy Change database, egy Manage Movie, és egy Recommendation System gombot, illetve az ezekhez tartozó eseménykezelő metódusokat (pl. a Table gomb esetében ShowTable\_Click), amik akkor futnak le, ha az adott gombra rákattint a felhasználó.

Az osztály tartalmazza a fő movies listát, és a TableInserts programrészek használatához szükséges változókat.

A Main osztály tartalmaz egy Loader nevű metódust, amely különböző metódushívásokat fog össze egy metódusban. Ennek a meghívására akkor van szükség, ha az adatbázishoz új film kerül hozzáadásra, filmet törlünk belőle, filmet módosítunk benne, vagy a teljes adatbázist cseréljük, amiből a program dolgozik.

Ez a metódus közel hasonló a Main osztály megnyitásakor lefutó Main\_Load metódushoz, viszont a legelején meghívjuk a korábban bemutatott LoadTableInserts metódust, amelyben a TableInsert programrész futtatásához szükséges metódusok vannak összegyűjtve. Erre azért van szükség, mivel ha az adatbázis filmjeinek bármilyen adata megváltozik, vagy az adatbázis cseréjekor azt feltételezzük hogy megváltozik a Movies táblában, akkor az összes többi táblát hozzá kell igazítani, végül pedig a filmeknek ezeket a változtatásaikat a programban is végre kell hajtani, szóval meghívni azokat ametódusokat is, amelyeket a Main\_Load-ban a program indulásakor tettünk.

Table gomb

A Table gomb a ShowTable elnevezést kapta a programon belül, így a ShowTable\_Click metódus fog a lenyomásának hatására lefutni.

/// <summary>

/// A Table gomb megnyomásakor lefutó metódus

/// </summary>

private void ShowTable\_Click(object sender, EventArgs e)

{

ShowTable showTable = new ShowTable(movies);

showTable.Show();

}

A metódusban példányosításra kerül a ShowTable osztály, illetve a létrehozott ablak láthatóvá is válik.

A ShowTable osztály konstruktorának segítségével átadjuk az osztálynak a movies listát, hiszen annak a tartalmát szeretnénk vele megjeleníteni. Az osztály egyetlen adattagot tartalmaz, a moviesToShow Movie típusú listát, amelyet az osztály konstruktorában egyenlővé teszünk a Main osztályban átadott movies listával.

Az osztály egyetlen metódusa a ShowTable\_Load metódus, amely az ablak megjelenítésekor fut le. Létrehozunk benne egy dataGridView példányt, amely adatforrása a konstruktorban értéket kapó moviesToShow lesz, tehát lényegében a Main osztály movies listája, amiben a filmek adatait tároljuk. Ezen lista tartalma kerül tehát megjelenítésre a Table gomb megnyomásával.

Change Database gomb

A Change Database gomb megnyomásával a ChangeDB\_Click metódus fut le. Lefutásakor a Windows Intéző tallózó ablaka nyílik meg, amely segítségével betallózhatunk egy másik, viszont fontos, hogy csak a movies.db szerkezetével teljes mértékben megegyező adatbázis fájlt. A betallózással megkapjuk a betallózandó fájl elérési útvonalát. A fájlt átmásoljuk a program saját könyvtárába movies.db fájlnévvel, tehát az eddigi adatbázis fájlt fogjuk vele kicserélni. A betallózás és csere után az adatbázis tábláinak, illetve a filmek listájának újratöltését a már korábban bemutatott Loader metódus végzi. A metódus asszinkron módban fut, így ez alatt az idő alatt a program megjelenít a felhasználó számára egy progress bar-t, hogy értesüljön arról, hogy az adatbázis frissítése történik.

ProgressBar osztály

A ProgressBar is egy grafikus ablakot megjelenítő osztály, amelynek a megjelenítésére az adatbázis frissítésekor kerül sor. Az osztály a konstruktoron kívül nem tartalmaz egy metódsut, vagy adattagot sem.

Grafikus elemeket tekintve tartalmaz egy Marquee stílusú progress bar-t, illetve egy „Updating the database, please wait…” feliratú label-t a felhasználó tájékoztatása céljából.

Manage Movie gomb

A Main metódus 3. gombja a Manage Movies gomb, a megnyomásának hatására fut le a ManageMoviesButton\_Click metódus. Ebben történik a ManageMovie osztály példányosítása a movies lista átadásával, és az így létrejött ablak megjelenítése.

A ManageMovie osztály segítségével kezelhetjük az adatbázisunkat a programon belül. Hozzáadhatunk új filmeket az adatbázisunkhoz, törölhetünk belőle filmeket, vagy akár módosíthatjuk is azoknak a különböző tulajdonságait.

Az ablak egy label segítségével megjelenít egy kérdést, amely arra vonatkozik, hogy a felhasználó melyik filmet szeretné módosítani. A label mellett helyezkedik el közvetlenül egy cboxTitle nevű legördülő lista (combobox), ebből tudja a felhasználó kiválasztani az adott filmet. Az ablak tartalmaz 15 darab textboxot, tehát a filmek minden tulajdonságához egyet, amelyekben a kiválasztott film összes adata megjelenítésre kerül a kiválasztás után.

Az ablak főképernyőjén 2 gomb található: bal oldalt egy Delete, ami a kiválasztott film törlésére szolgál, jobb oldalt pedig egy Modify, amit ha megnyomunk azután, hogy a kiválasztott film adatait módosítottuk igényeink szerint a textboxokban, akkor módosítja őket az adatbázisban, hozzáigazítja a különböző táblák adatait, és a program filmeket tartalmazó listáját is. Jobb felül, a legördülő lista mellett található egy Add New gomb, amely új film beszúrására ad lehetőséget. Ezt megnyomva eltűnik maga a gomb, a legördűlő lista, a Delete és a Modify gombok is, a textboxokból pedig töröljük a benne lévő szövegeket. Az eltűnt gombok helyén 2 új gombot jelenítünk meg: a Back gombot, amivel a filmek módosítását és törlését megvalósító képernyőre tudunk visszamenni, illetve az Add gombot, amit megnyomva miután a felhasználó a textboxokat feltöltötte az általa hozzáadni kívánt film adataival, a program hozzáadja azt a filmeket tartalmazó listához, és az adatbázist is hozzá igazítja. Az Add gomb megnyomása után visszatér a program az előző képernyőre, ahol a filmek adatait tudjuk módosítani, törölni.

A ManageMovie osztály 3 adattagot tartalmaz: a Movies típusú moviesForModify listát, a konstruktorban ezt egyenlővé tesszük a Main osztály movies listájával amiben a filmek adatait tároljuk. A Movie típusú selectedMovie adattagot, ebben fogjuk eltárolni a módosítandó film adatait. Végül pedig a bool típusú inputError adattagot, amelynek értékét a beviteli hibák kiszűrésére használjuk fel.

Az ablak megnyílásának eseményére a ManageMovie osztály ManageMovie\_Load metódusa fut le. Ebben egy egyszerű metódushívás történik, a SetCboxDetails metódust hívja meg. Ez a metódus az ablak működésének alapját adó combobox tulajdonságait állítja be. A comboboxot lenyitva a filmek címei jelennek meg, és az Id-jük segítségével azonosítjuk őket.

/// <summary>

/// Combobox tulajdonságainak beállítása

/// </summary>

private void SetCboxDetails()

{

cboxTitle.DataSource = moviesForModify; cboxTitle.DisplayMember = "Title";

cboxTitle.ValueMember = "Id";

}

Ha a combobox, azaz a legördülő listában megváltozik a választott film, akkor a cboxTitle\_SelectedIndexChanged metódus fut le. Ennek a metódusnak a tartalma szintén egy egyszerű metódushívás, a FillupTextboxes metódust hívja meg. Ez a metódus az elején eltárolja a combobox segítségével választott filmet a Movie típusú selectedMovie változóban, a textboxokat pedig ennek a változónak a segítségével feltölti a legördülő listából kiválasztott film megfelelő adataival.

A korábban már említett Modify gombot megnyomva a buttonModify\_Click kerül meghívásra. Miután az IsEmpty metódussal meggyőződött róla, hogy egyik textbox sem maradt üresen, a selectedMovie adatait frissíti a textboxokban megadottakra. Mivel a selectedMovie-nak referencia szerint adtuk át a kiválasztott filmet, és nem egy másolatot készítettünk róla, ezért a selectedMovie módosításával a film az eredeti listában is frissül. A GetFromTextboxes metódust felhasználva a textboxokból kiolvassa a program az adatokat, és amennyiben nincs input hiba, akkor az adatbázisban is frissíti az adott film adatait a selectedMovie változó adatait felhasználva. A frissítéshez az SqlConnector UpdateCommand metódusát használja fel, ez hajtja végre az összekészített SQL parancsot. Ezután a már bemutatott Loader segítségével frissítjük, újratöltjük az adatbázis többi tábláját és a movies listát az id-kat tartalmazó segédadattagok frissítésének érdekében. A Loader metódus futása alatt megjelenítjük a korábban már bemutatott progress bar-t, így ez alkalommal is szükség van az asszinkron módban futtatásra.

/// <summary>

/// A Modify gomb hatására végrehajtandó algoritmus

/// </summary>

private async void buttonModify\_Click(object sender, EventArgs e)

{

if (IsEmpty() == false)

{

GetFromTextboxes(ref selectedMovie);

if (inputError == false)

{

SqlConnector conn = new SqlConnector(Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "movies.db"));

string command;

command = string.Format("UPDATE Movies SET Title= \"{0}\", Genre= \"{1}\", Released= \"{2}\", Runtime= {3}, Gender\_of\_the\_protagonist={4}, Main\_actor=\"{5}\", Keywords=\"{6}\", Director=\"{7}\", Language=\"{8}\", Production\_countries=\"{9}\", Tmdb\_score={10}, Number\_of\_ratings={11}, Popularity={12}, Budget={13}, Revenue={14} WHERE ID={15}",

selectedMovie.Title, selectedMovie.GenreStringWithCommas, selectedMovie.Released, selectedMovie.Runtime, selectedMovie.GenderOfProtagonist, selectedMovie.MainActor, selectedMovie.KeywordStringWithCommas, selectedMovie.DirectorStringWithCommas, selectedMovie.LanguageStringWithCommas, selectedMovie.CountryStringWithCommas, selectedMovie.TmdbScore.ToString(System.Globalization.CultureInfo.InvariantCulture), selectedMovie.NumberOfRatings, selectedMovie.Popularity.ToString(System.Globalization.CultureInfo.InvariantCulture), selectedMovie.Budget, selectedMovie.Revenue, selectedMovie.Id);

conn.UpdateCommand(command);

Main main = new Main();

ProgressBar progressBar = new ProgressBar();

progressBar.Show();

try

{

await Task.Run(() => main.Loader());

finally

{

progressBar.Visible = false;

}

}

}

}

A buttonModify\_Click metódushoz kissebb változásokkal, de alapvetően hasonlóak a Delete gomb megnyomásakor lefutó buttonDelete\_Click és az Add gomb megnyomásakor lefutó buttonAdd\_Click metódusok is. Pl. a hozzáadás esetében nem frissítő SQL parancsot készítünk össze hanem beszúrót, a törlés esetén pedig nem szükséges a textboxok ürességének ellenőrzése, hiszen nem használjuk azokat a törléshez, illetve a listánkban az adott filmet nem módosítjuk, hanem töröljük, vagy hozzáadjuk.

Az új film beszúrásához szükséges felület megjelenítéséért az Add New gomb felel. Ennek megnyomására lefut a buttonAddNew\_Click metódus, amely kiüríti a textboxokat, megjeleníti és elrejti a szükséges grafikus elemeket, gombokat.

/// <summary>

/// A módosító felület átalakítása új film beszúrásához

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void buttonAddNew\_Click(object sender, EventArgs e)

{

cboxTitle.Visible = false;

buttonAdd.Visible = true;

buttonAddNew.Visible = false;

buttonDelete.Visible = false;

buttonModify.Visible = false;

buttonBack.Visible = true;

labelQuestion.Visible = false;

labelAddNew.Visible = true;

textBoxTitle.Text = String.Empty;

.

.

.

textBoxRevenue.Text = String.Empty;

}

Az algoritmusok működéséhez segítségül szolgálnak, hasonlóképpen mint a film módosításánál a GetFromTextboxes és az IsEmpty metódusok. A GetFromTextboxes segítségével kiolvassuk a textboxok tartalmait és az aktuális adattag típusára alakítjuk, amelyben tárolni fogjuk, az IsEmptiy metódussal pedig leellenőrizzük hogy nem-e maradt üresen textbox. Ha maradt üresen akkor igaz értéket ad vissza, ha nem, akkor pedig hamisat.

/// <summary>

/// Az adatok kiolvasása a textboxokból

/// </summary>

/// <param name="selectedMovie">A kiolvasott adatokat a selectedMovie-ban tároljuk el</param>

public void GetFromTextboxes(ref Movie selectedMovie)

{

inputError = false;

if (IsEmpty() == false)

{

try

{

selectedMovie.Title = textBoxTitle.Text;

selectedMovie.GenreStringWithCommas = textBoxGenre.Text;

selectedMovie.Released = int.Parse(textBoxReleased.Text);

.

.

.

selectedMovie.Budget = long.Parse(textBoxBudget.Text);

selectedMovie.Revenue = long.Parse(textBoxRevenue.Text);

}

catch

{

MessageBox.Show("Check the input data!", "Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

inputError= true;

}

}

}

A megjelenített textboxokra mozgatva az egeret a speciális formázást igényló mezőknél a program egy label segítségével nyújt útmutatást a felhasználó számára. A label megjelenik, ha az egér a textboxon áll, eltűnik, ha elhagyja a textboxot.

/// <summary>

/// Egy címke megjelenítése ha az egér a textBoxGenre-en áll

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void textBoxGenre\_MouseEnter(object sender, EventArgs e)

{

labelGenreHint.Visible = true;

}

Az osztály tartalmaz még egy HomeScreen nevű metódust is, amely metódushívásokat foglal össze abból a célból, hogy az ablakot vissza tudja állítani olyan állapotba, mint amikor rányomott a felhasználó a Manage Movies gombra. A metódus a Back gomb lenyomására lefutó buttonBack\_Click és az Add gomb megnyomásakor lefutó buttonAdd\_Click metódusokban kerül meghívásra.

/// <summary>

/// A ManageMovie osztály ablakának visszaállítása megnyitáskori állapotára

/// </summary>

public void HomeScreen()

{

SetCboxDetails();

FillupTextboxes();

cboxTitle.Visible = true;

buttonModify.Visible = true;

buttonBack.Visible = false;

buttonDelete.Visible = true;

buttonAddNew.Visible = true;

buttonAdd.Visible = false;

labelQuestion.Visible = true;

labelAddNew.Visible = false;

}

Becslésekre vonatkozó osztályok (DecTreeForLanguages, DecTreeForGender, DecTreeForDirector, DecTreeForTmdb)

A programom fő célja ugyan az, hogy a felhasználónak az általa megválaszolt kérdéseket figyelembe véve egy olyan filmet tudjon ajánlani, ami egyezik az érdeklődési körével, igényeivel. Viszont a becslésekhez használatos metódusok, illetve magának az osztályozó modellnek a felépítésének megismeréséhez, illetve begyakorlásához kezdetben pár egyszerűbb osztályozási problémát vizsgáltam: megpróbáltam megbecsülni a filmek különböző tulajdonságainak figyelembevételével a filmek nyelvét, a főszereplő nemét, a rendező nevét, illetve az adott film Tmdb pontszámát.

A program ezek közül minden egyes osztályozási problémához tartalmaz egy osztályt, amelyben az osztályozó modell betanítása, továbbá a modellt felhasználva maga a becslés is történik. Ezekhez az osztályokhoz kapcsolódóan két másik osztályt is tartalmaz a program: egyikben az adott becsléshez felhasznált adattagjai szerepelnek a filmeknek, a másikban pedig az az adattag, amelynek megbecslését el szeretnénk végezni.

A fent felsorolt becslések programkódjai nem sokban különböznek egymástól, ezáltal én most csak közülük egynek a működését, felépítését fogom ismertetni, a Tmdb pontszám megbecslésére irányuló DecTreeForTmdb osztályt, illetve a hozzá kapcsolódó 2 osztályt.

**Tmdb osztály**

internal class Tmdb

{

public int[] Keyword { get; set; }

public int[] Genre { get; set; }

public double TmdbScore { get; set; }

Azokat az adattagokat tartalmazza, amelyek a modell betanításához szükségesek lesznek. Jelen esetben a kulcsszavak és a műfajok alapján történik a becslés, és a Tmdb pontszámra irányul, ezért a TmdbScore adattagra is szükség van, mivel ebben tároljuk el a betanítási adatok pontszámait.

**TmdbPredict osztály**

internal class TmdbPredict

{

[ColumnName("Score")]

public float PredictedTmdb { get; set; }

}

Egyetlen adattagot tartalmaz, az adott film Tmdb pontszámát, hiszen erre irányul a becslés.

**DecTreeForTmdb**

Ez az az osztály, ahol a tanító adathalmaz megadása történik az adatbázis első 20 filmjének felhasználásával, a modell felépítése, betanítása, az adott film kiválasztása, aminek a pontszámát szeretnénk megbecsülni, illetve maga a becslés is. A becsléshez egyéni séma definiálása is szükséges annak érdekében, hogy dinamikus méretűek legyenek a vektorok, tehát ne okozzon hibát az algoritmusban ha az adatbázis több/kevesebb filmet tartalmaz, így ezáltal pl. a kulcsszavak, műfajok száma is változik, hanem alkalmazkodni tudjon hozzá. Az ML.NET bemutatásánál már ismertetve lett, hogy az adatokon átalakítást kell végezni, így ezek természetesen ennél a becslésnél is szükségesek. A felhasznált tanító adatoknál szükséges az átalakítás, hiszen a prop.GenreContains és a prop.KeywordContains adattagok tartalma, amikből kiolvassuk a filmekhez tartozó megfelelő adatokat 0 vagy 1 lehet csak, ezáltal int típusú tömbök. Ezeket a kiolvasott int típusú adatokat a metódusban átalakítjuk Single-re, ami a C#-ban gyakorlatilag float típust jelent.

A becsléshez, mivel a Tmdb pontszám egy folytonos érték, így egy regressziót használó Fasttree került felhasználásra, annak paramétereinek beállításával. Az osztály egyetlen metódust tartalmaz, az alábbi BuildTree metódust:

public void BuildTree(List<Movie> moviesL, PropertiesForDecTree prop)

{

var mlContext = new MLContext();

var movies = new List<Tmdb>();

for (int i = 20; i < moviesL.Count; i++)

{

Tmdb newTmdb = new Tmdb

{

Genre = prop.GenreContains[i],

Keyword = prop.KeywordContains[i],

TmdbScore = (float)moviesL[i].TmdbScore

};

movies.Add(newTmdb);

}

var schemaDef = SchemaDefinition.Create(typeof(Tmdb));

schemaDef["Keyword"].ColumnType = new VectorDataViewType(NumberDataViewType.Int32, prop.KeywordAll.Count);

schemaDef["Genre"].ColumnType = new VectorDataViewType(NumberDataViewType.Int32, prop.GenreAll.Count);

var data = mlContext.Data.LoadFromEnumerable(movies, schemaDef);

var dataProcessPipeline = mlContext.Transforms.Conversion.ConvertType(nameof(Tmdb.TmdbScore), nameof(Tmdb.TmdbScore), DataKind.Single)

.Append(mlContext.Transforms.Conversion.ConvertType("KeywordEncoded", nameof(Tmdb.Keyword), DataKind.Single))

.Append(mlContext.Transforms.Conversion.ConvertType("GenreFloat", nameof(Tmdb.Genre), DataKind.Single))

.Append(mlContext.Transforms.Concatenate("Features", "KeywordEncoded", "GenreFloat"))

.AppendCacheCheckpoint(mlContext);

var trainer = mlContext.Regression.Trainers.FastTree(

labelColumnName: nameof(Tmdb.TmdbScore),

featureColumnName: "Features",

numberOfLeaves: 20,

numberOfTrees: 150,

minimumExampleCountPerLeaf: 8

);

var trainingPipeline = dataProcessPipeline.Append(trainer);

var model = trainingPipeline.Fit(data);

var predictions = model.Transform(data);

var metrics = mlContext.Regression.Evaluate(predictions, labelColumnName: nameof(Tmdb.TmdbScore));

Console.WriteLine($"R^2: {metrics.RSquared}");

Console.WriteLine($"Mean Absolute Error: {metrics.MeanAbsoluteError}");

Console.WriteLine($"Mean Squared Error: {metrics.MeanSquaredError}");

var predictionEngine = mlContext.Model.CreatePredictionEngine<Tmdb, TmdbPredict>(model, inputSchemaDefinition: schemaDef);

int film = 19;

Console.WriteLine(moviesL[film].Title);

var testMovie = new Tmdb

{

Genre = prop.GenreContains[film],

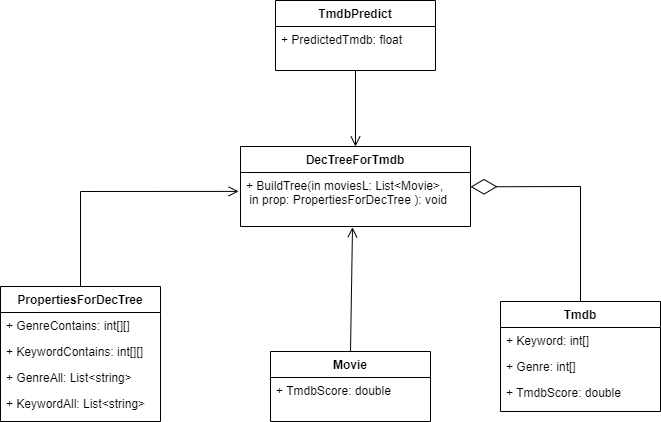
Keyword = prop.KeywordContains[film]

};

var prediction = predictionEngine.Predict(testMovie);

Console.WriteLine($"Predicted TmdbScore: {prediction.PredictedTmdb}");

}



A Tmdb pontszám becslésére vonatkozó algoritmus annyiban eltér a korábban említett többi becsléstől, hogy ez az algoritmus nem tartalmazza az egész adathalmazra nézve a hatékonyság mérését, mivel erre a MeasureAccuracy osztályban kerül majd sor a későbbiekben, ahol a különböző becslési algoritmusok pontosságának az összevetése történik.

A film nyelvére vonatkozó döntési fa hatékonysága, illetve a hatékonyságának mérési módszere a kézzel készített döntési fával összevetve már bemutatásra került. Ugyanezzel a módszerrel került lemérésre a következő 2 becslésnek a hatékonysága is, amely a film rendezőjére, és a főszereplő nemére vonatkozott.

A film rendezőjére vonatkozó döntési fánál a következő hatékonysági számokat kaptam:

|  |  |  |  |
| --- | --- | --- | --- |
| Intervallum | "1-20" | "60-80" | Összesen |
| Helyes | 15 | 18 | 33 |
| Összesen | 20 | 20 | 40 |
| Hatékonyság | 75,00% | 90,00% | 82,50% |

**x**. táblázat: A film főszereplőjének nemére vonatkozó becslő algoritmus mérési eredményei

A mérési adatokból elsőre azt a következtetést szűrhetnénk le, hogy rendkívül pontosan működik az algoritmus, viszont ez valójában nem igaz. A filmek egész adathalmaza eléggé kiegyensúlyozatlan a főszereplők nemét illetően, a tanítóhalmazként szolgáló utolsó 20 film esetében csak 3 filmnél volt a főszereplő neme nő. Ebből adódóan az algoritmus mindkét vizsgált tartományban az összes filmre férfi főszereplőt becsült, és mivel az egész adathalmazt nézve rendkívül kicsi a női főszereplők aránya, ezért ez valóban nagy hatékonysághoz vezetett. A 2 tartomány filmjeit vizsgálva összességében 82,5%-os hatékonysággal tudott működni az algoritmus.

A filmek rendezőjére vonatkozó becslés hatékonyságának mérése során a következő eredményeket kaptam:

|  |  |  |  |
| --- | --- | --- | --- |
| Intervallum | "1-20" | "60-80" | Összesen |
| Helyes | 0 | 1 | 1 |
| Összesen | 20 | 20 | 40 |
| Hatékonyság | 0,00% | 0,05% | 0,025% |

**x**. táblázat: A film rendezőjére vonatkozó becslő algoritmus mérési eredményei

A táblázat alapján azt a következtetést vonhatjuk le, hogy az algoritmus egyáltalán nem működött jól, hiszen 40 alkalomból csupán 1 alkalommal sikerült eltalálnia helyesen a film rendezőjét. Ennek okai, hogy a film rendezői eléggé változatosak az adathalmazban, kevés az ismétlődő név, illetve az algoritmus eleve csak azokból a nevekből tudott választani, amelyek szerepeltek a 20 elemű mintahalmazban. Ezt a két dolgot figyelembe véve pedig érthető, hogy az algoritmusnak nem sikerült olyan mintákat találnia, amik által helyesen meg lehetett volna becsülni a filmek rendezőinek a nevét.

**MeasureAccuracy osztály**

A MeasureAccuracy osztály az ML.NET által támogatott, regresszióhoz használható becslési algoritmusok pontosságának összehasonlításához került létrehozásra. Az osztályban 5 algoritmus, a Fasttree, Sdca, LbfgsPoissonRegression, Gam és OnlineGradientDescent pontosságai kerülnek kiszámításra. A különböző becslések alapját a DecTreeForTmdb osztály BuildTree algoritmusa adta, mivel a becslések az összehasonlítás esetében is arra irányultak, hogy egy 20 elemű tanítóhalmaz alapján melyik algoritmus tudja a legpontosabban megbecsülni a 100 db film Tmdb pontszámait.

Az osztály tartalmazza az 5 algoritmusnak a finomhangolt, és a finomhangolás nélküli verzióját is. A finomhangolás nélküli algoritmusok felépítése annyiban tér el a DecTreeForTmdb BuildTree metódusától, hogy a Fasttree trainer helyett értelemszerűen a vizsgált becslő algoritmust adjuk meg trainernek finomhangolható paraméterek használata nélkül, a becslést pedig nem csak egy filmre, hanem az összesre elvégezzük, és a kapott eredményeket eltároljuk egy listában. Az algoritmus végén pedig kiszámítjuk az R^2 mérőszámot a hatékonyság mérése céljából, viszont nem a beépített algoritmussal, mint a BuildTree-ben hanem saját algoritmussal, ami a mérést nem csak a tanító halmazra, hanem a teljes adathalmazra végzi.

A finomhangolt verzióban is megtalálhatóak ezek a változtatások, viszont itt különböző paramétereket használunk a trainerhez finomhangolás céljából. Ezek a paraméterek például a Fasttree esetében a következőek: numberOfLeaves, tehát a fa leveleinek száma, a numberOfTrees, tehát a fák száma, a minimumExampleCountPerLeaf, tehát a minimális példák levelenként, és a learningRate, tehát a tanulási ráta.

var trainer = mlContext.Regression.Trainers.FastTree(

labelColumnName: nameof(Tmdb.TmdbScore),

featureColumnName: "Features",

numberOfLeaves: leaves,

numberOfTrees: trees,

minimumExampleCountPerLeaf: minExample,

learningRate: learningRate

);

Ezeknél a paramétereknél az optimális értékeket a tanító halmaz befolyásolja, tehát akár 1 film adatainak a megváltoztatása esetén is elképzelhető, hogy már más lesz az optimális érték valamelyik paraméter értéke esetén.

Ezért a paraméterek legjobb hatékonyságot adó értékeinek megtalálásához bizonyos tartományokon belül véletlen számokat generálunk, és R^2 szempontjából kiértékeljük a modellt. Ezt a műveletet mindegyik algoritmus esetében 200 alkalommal elvégezzük, és ha az adott értékekkel magasabb R^2 értéket kapunk, akkor a korábbi eltárolt legjobbat lecseréljük az aktuálisra.

.

.

.

int leaves = random.Next(2, 10);

int trees = random.Next(20, 200);

int minExample = random.Next(3, 10);

double learningRate = random.NextDouble() \* (0.2 - 0.01) + 0.01;

.

.

.

if (bestRSquared < RSquare(moviesL, predictedTmdbs))

{

bestRSquared = RSquare(moviesL, predictedTmdbs);

}

.

.

.

A modellek pontosságának kiértékelését az RSquare metódus végzi. A metódus az R^2 érték számításához használt formula segítségével kiszámolja a pontosságot a paraméterként kapott becsült Tmdb pontszámaival, és a filmek valós Tmdb pontszámaival végzett számítások során. A metódus visszatérési értéke a float típusú RSq váltózó, ez tartalmazza a kiszámolt R^2 értéket.

Az osztályban található 10 darab metódus hívását a MeasureAll metódus fogja össze, így ennek az egynek a meghívásával lemérhető az összes algoritmus hatékonysága.

Recommendation System gomb

**SetPriority osztály**

A fő becslés elkészítése előtt a felhasználótól adatokat kell bekérnünk kérdésekre adott válaszok formájában annak érdekében, hogy megismerjük az ízlését a filmekre vonatkozóan. Mivel az emberek számára más és más jellemzők fontosak a filmek esetében is, ezért a kérdések között olyan jellemzőre vonatkozó is előfordulhat, amely az aktuális felhasználó számára kevésbé, vagy egyáltalán nem fontos.

Ennek az esetnek az elkerülését azzal előzhetjük meg, hogy a kérdések megjelenítése előtt megkérjük a felhasználót arra, hogy válassza ki a film jellemzői közül a számára fontos jellemzőket, és rangsorolja őket prioritás szerint. A SetPriority grafikus osztály ezt a feladatot valósítja meg.

Az osztály működésének koncepciója, hogy létrehozunk 2 PriorityListItem típusú listát. A PriorityListItem osztályban a ToString() metódust felülírjuk annak érdekében, hogy ha később hivatkozunk egy PriorityListItem lista egy-egy elemére, akkor a Title tulajdonsága alapján beazonosítható legyen.

A PriorityList osztály a következő képpen néz ki:

internal class PriorityListItem

{

public int Id { get; set; }

public string Title { get; set; }

public int Priority { get; set; }

public override string ToString() {

return Title;

}

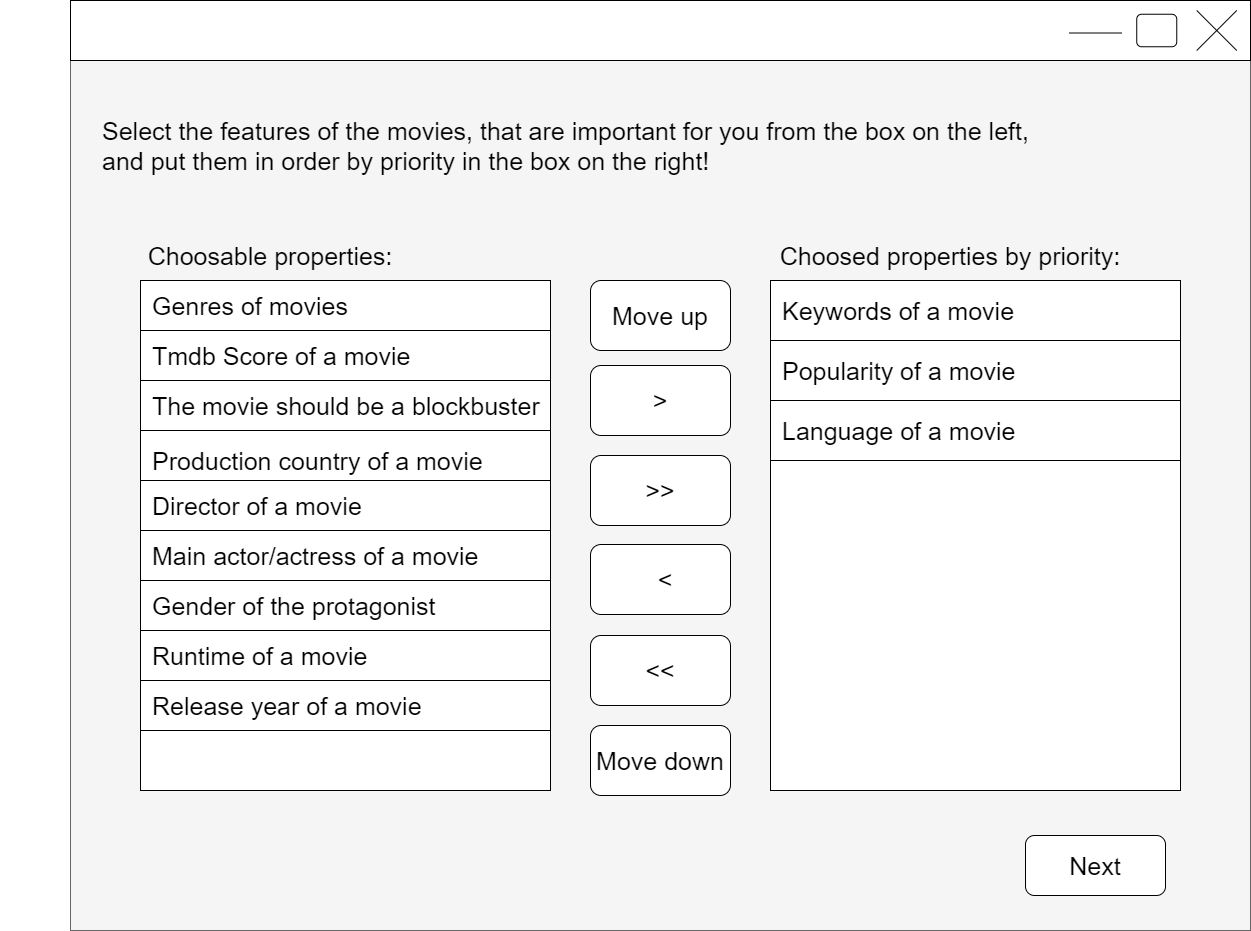
}

Létrehozunk egy enumot is a választható tulajdonságokkal, amit majd később a kérdéseknél is használni fogunk annak érdekében, hogy össze lehessen kapcsolni a kiválasztott és prioritási sorrendbe tett jellemzőket az ahhoz kapcsolódó kérdésekkel, mégpedig úgy, hogy a kérdéseknek is lesz olyan adattagja, amelyet beállítunk arra az enum értékre, amely jellemzőhöz kapcsolódik a kérdés.

A 2 létrehozott lista az originalList és a selectedList. A selectedList üresen marad, az originalList-et pedig feltöltjük a következő módon:

* Az Id adattagok az enum elemei lesznek int-re castolva
* A Title az adott jellemző nevét jelöli
* A Priority érték mindenhol 0 értéket vesz fel, hiszen a prioritás ebben a listában még lényegtelen.

Az osztályhoz tartozó ablak a következő képpen néz ki:



A bal oldali listbox-ban található az originalList lista tartalma, ezekből a jellemzőkből válogathatja ki a felhasználó a számára fontos jellemzőket. A jobb oldali listbox a selectedList elemeit tartalmazza, tehát kezdetben üres.

Az originalList-ből a „>” és a „>>” gombokkal lehet a kiválasztott jellemzőt/jellemzőket átmozgatni a selectedList-be, visszafelé pedig a „<” és „<<” gombokkal attól függően, hogy csak 1 darab jellemzőt szeretnénk átmozgatni, vagy az összeset. Egy jellemző egyszerre csak egy listában lehet jelen, tehát átmozgatás után mind a két listához tartozó listboxot frissíteni kell, hogy az egyik helyen eltűnjön, a másik helyen pedig megjelenjen az átmozgatott tulajdonság.

A „Move Up” és „Move Down” gombokkal a jobb oldali listbox tartalmát tudjuk módosítani miután kerültek bele elemek. Ezekkel a gombokkal mozgathatjuk felfelé és lefelé a kiválasztott jellemzőt, tehát a prioritási értékét módosíthatjuk. A legfelül lévő elem rendelkezik a legnagyobb prioritással, a legalján lévő elem pedig a legkisebbel.

A „>” gomb algoritmusa a következő:

private void MoveSelectedItem()

{

if (listBoxOriginal.SelectedItem != null) {

PriorityListItem selectedItem = (PriorityListItem)listBoxOriginal.SelectedItem; originalList.Remove(selectedItem); //Az elem törlése korábbi helyéről

selectedList.Add(selectedItem); //Az elem hozzáadása az új listához

RefreshListBoxes(); //A listák tartalmait megjelenítő listboxok frissítése a változtatások megjelenítéséhez

}

}

A többi elemmozgatásra használt gomb algoritmusa is ezen a logikán alapul. Ezeknek az algoritmusoknak a végén megjelenik a RefreshListBoxes metódus hívása, amely a 2 listbox adatforrását először kinullázza, majd újból beállítja a 2 listára annak érdekében, hogy a listboxok a listák naprakész állapotát tudják megjeleníteni. Ennek a metódusnak a végén az UpdatePriorities is megjelenik:

/// <summary>

/// Módosítások után az elemek prioritásainak frissítése a sorrend alapján

/// </summary>

private void UpdatePriorities()

{

for (int i = 0; i < selectedList.Count; i++)

{

selectedList[i].Priority = i; // Minél előrébb (a listboxban fentebb) van egy elem, annál alacsonyabb értéket kap, ami pedig magasabb prioritást jelent

}

}

A „Move Up” és a „Move Down” gombok a következő logika szerint működnek:

/// <summary>

/// A kiválasztott elem felfelé mozgatása a listboxban a magasabb prioritás érdekében

/// </summary>

private void MoveUp()

{

int selectedIndex = listBoxSelected.SelectedIndex; //A kiválasztott elem indexét eltároljuk

if (selectedIndex > 0) //Ellenőrizzük hogy nem-e 0, hiszen ha igen, akkor az azt jelenti hogy a lista tetején van, tehát nem lehet fentebb mozgatni

{

var item = selectedList[selectedIndex]; //Egy átmeneti változóban tároljuk a mozgatni kívánt elemet

selectedList.RemoveAt(selectedIndex); //A listából töröljük az elemet a korábbi helyéről

selectedList.Insert(selectedIndex - 1, item); //Beszúrjuk az átmenetileg eltárolt elemet eggyel kisebb index használatábal, így a listboxban eggyel feljebb kerül

RefreshListBoxes(); // A listboxok frissítése

listBoxSelected.SelectedIndex = selectedIndex - 1; // Miután az elem megjelent eggyel feljebb a listboxban, átvisszük rá a kijelölést is, hogy ne az alatta lévő elem maradjon kijelölve.

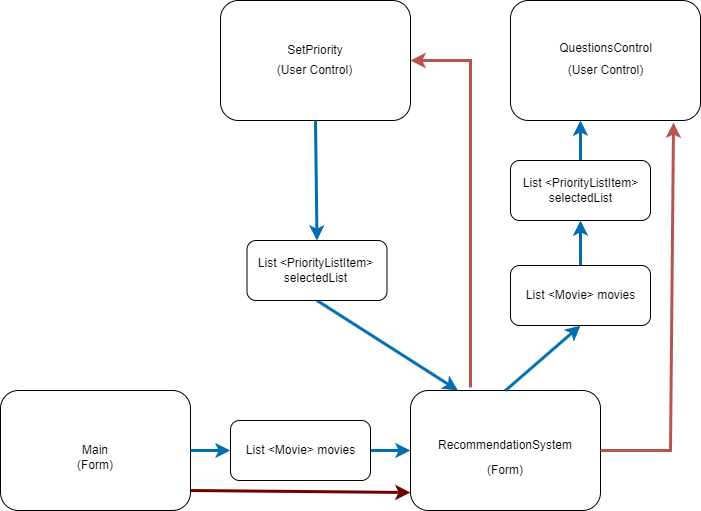
}

}

**QuestionsControl osztály**

A különböző jellemzők prioritásának felmérése után az ablak jobb alsó sarkában elhelyezett „Next” gombra kattintva a felhasználó tovább léphet a kérdésekre. A kérdések megjelenítése, és a rájuk adott válaszok kezelése a QuestionsControl elnevezésű osztályban valósul meg. A SetPriority és a QuestionsControl, majd később a Results osztályok szorosan összekapcsolódnak, külön-külön egyiküknek sincs értelme, ezért UserControl-ként kerültek létrehozásra annak érdekében hogy a tartalmukat ugyanabban a Form ablakban lehessen megjeleníteni. Ez az ablak és egyben osztály a RecommendationSystem, ebben van megvalósítva az, hogy alapértelmezetten a SetPriority osztály grafikus felületének tartalma töltődjön be, majd a „Next” gomb megnyomására eseménykezelő használatával a SetPriority helyett megjelenítjük a QuestionsControl osztály grafikus tartalmát, végül pedig az utolsó kérdés feltétele után a finish gombot, amely megnyomására megjelenik a Results osztály tartalma az ajánlott filmekkel együtt.

Az osztálynak szüksége van a SetPriority osztályban beállított prioritást tartalmazó listára, hiszen a kérdéseket ez alapján tesszük fel, illetve a Main osztály movies listájára is, ami a filmek adatait tartalmazza, hiszen azokból alakítjuk ki a tanítóhalmazt, amelynek adatait felhasználjuk a kérdésekhez tartozó válaszlehetőségeknél. Továbbá a kérdésekre adott válaszokkal alakítjuk ki a tanítóhalmaz pontszámait, amik felhasználásával tudja majd a program megbecsülni ezt a pontszámot a teljes adathalmaz filmjeire, a legmagasabb pontszámokat kapott filmeket pedig megjeleníteni a felhasználó számára, mint ajánlott filmek.



Az ábra megmutatja, hogy melyik osztály grafikus felülete honnan kerül megjelenítésre, illetve milyen listák kerülnek köztük átadásra, és honnan. Az objektumok átadását a kék nyilak jelölik, a megjelenítést a vörös színű nyilak.

Mint már említésre került, a QuestionsControl-ban kerülnek létrehozásra a kérdések, ezek köré, illetve ezen kérdésekre adott válaszok köré épül az egész osztály. A kérdésekhez létrehozásra került egy Question osztály, amelynek két adattagja van, a string típusú Title, amiben a kérdéseknek a szövege kerül tárolásra, illetve a Type, amiben a SetPriority-ben is már használt Features enum típus megfelelő értékét fogjuk int-re castolva eltárolni, tehát gyakorlatilag ezzel azonosítjuk hogy az adott kérdés a filmek melyik jellemzőjére vonatkozik, és ezzel kapcsoljuk őket össze a prioritási listában kiválasztott elemekkel.

/// <summary>

/// Osztály létrehozása a kérdésekhez

/// </summary>

internal class Question

{

public string Title {get;set;}

public int Type {get;set;}

}

A QuestionsControl osztályban létrehozásra került a fenti osztály felhasználásával egy Question típusú questionsList lista, a CreateQuestions metódusban ezt a listát töltjük fel kérdésenként, minden egyes kérdés szövegének és típusának megadása után.

Összesen 12 darab jellemző közül választhat a felhasználó a SetPriority képernyőn, tehát kérdésekből is 12 darab van összesen.

A kérdéseket a válaszlehetőségek szerint alapvetően 3 típusra lehet osztani:

* Igaz/hamis, a megfelelő checkbox kiválasztásával
* A felhasználó kedvenc dolgainak kiválasztása a listából checkboxokkal, akár egyszerre többet is
* Érték/tartomány beállítása csúszkák (Trackbar) segítségével

/// <summary>

/// A kérdések összeállítása

/// </summary>

public void CreateQuestions()

{

Question temp = new Question

{

Title = "What is the lowest Tmdb score below which you will not watch a movie?\n\nPlease set the value with the slider, or write it into the box!",

Type = (int)Features.TmdbScore

};

questionsList.Add(temp);

temp = new Question

{

Title = "Which languages do you prefer in a movie?\n\nPlease select the languages with the checkboxes!",

Type = (int)Features.Language

};

questionsList.Add(temp);

temp = new Question

{

Title = "Does it important for you to be the movie a blockbuster?\n\nGive your answer by selecting the appropriate checkbox!",

Type = (int)Features.Blockbuster

};

questionsList.Add(temp);.

.

.

.

}

A kérdések összeállítása után a megjelenítésükről a LoadQuestion metódus gondoskodik. A SetPriority osztályból megkapott, a felhasználó által kiválasztott jellemzőket prioritási sorrendben tartalmazó prioritási listán végigmegy, és mivel a prioritási lista tagjainak Id-ja és a kérdések Type adattagja is egyaránt a Features enum megfelelő értékeit kapták meg, ezért egyezést keresve könnyen hozzárendelhető a prioritási lista egyes elemeihez a megfelelő kérdés. A prioritási listában való lépkedést az int típusú currentIndex osztály szintű változóval oldjuk meg, amelyet csak a Next gomb megnyomásakor, tehát a következő kérdésre történő tovább lépéskor növelünk meg.

/// <summary>

/// A kérdések megjelenítése a beállított prioritási lista segítségével

/// </summary>

public void LoadQuestion()

{

for (int i = 0; i < questionsList.Count; i++)

{

if (receivedPriorityList[currentIndex].Id == questionsList[i].Type)

{

labelQuestion.Text = questionsList[i].Title;

}

}

}

Ahhoz, hogy az összes filmet tartalmazó halmaztól el lehessen különíteni a tanítóhalmazt, egy olyan lista létrehozására van szükség, amely csak a tanítóhalmaz filmjeit és azok adatait tartalmazza. Ahhoz, hogy a tanítóhalmazra is meg lehessen valósítani egy hasonló becslést, mint a Tmdb pontszámnál, az egyes filmeknek tartalmazniuk kell a korábban az Algorithms osztályban feltöltött PropertiesForDecTree típusú properties-nek azon adattagjait, amelyek az adott filmek esetében az egyes, több értéket is felvehető jellemzőkhöz tartozó egyforma hosszúságú 1 és 0 értékeket tartalmazó tömböket tartalmazzák, hiszen az ML.NET ilyen formában tudja ezeket az adatokat kezelni.

Mivel ezáltal a tanítóhalmaz filmjeinek adattagjai el fognak térni az eredeti Movie típusú movies listáétól, így egy külön osztály került létrehozásra, ami a LearningMovie nevet kapta. A Movie osztályhoz hasonlóan tartalmazza a filmek azon egyszerű tulajdonságait tároló adattagokat, amelyeket majd a becsléshez fogunk használni, ezek a Released-től kezdve a Score-ig megjelenő adattagok. De a Movie osztállyal ellentétben a több értéket is felvehető tulajdonságokhoz, tehát a properties adatainak tárolásához is tartalmaz tömböket (Genre-től a ProductionCountry-ig). Ezek mellett szöveges formában is el vannak tárolva filmenként ezek az adatok (Genrestring-től CountryString-ig), hiszen ezen adatokkal vetjük majd össze a felhasználó válaszait, amik alapján pedig majd eldöntjük, hogy igaz-e az adott válasz az adott filmre, tehát növeljük-e a pontszámát, vagy sem. Viszont a későbbiekben a becsléshez ezek az adatok nem használhatóak, ezért ebből a célból el kell tárolni a tömbökben a properties-nek az adott filmhez tartozó tömbjeit is a becsléshez.

/// <summary>

/// A tanítóhalmaz filmjeihez létrehozott típus

/// </summary>

public class LearningMovie

{

public int Released { get; set; }

public int Runtime { get; set; }

public int GenderOfProtagonist { get; set; }

public string MainActor { get; set; }

public double TmdbScore { get; set; }

public bool IsPopular { get; set; }

public bool Blockbuster { get; set; }

public double Score { get; set; }

public int[] Genre { get; set; }

public int[] Keyword { get; set; }

public int[] Director { get; set; }

public int[] Language { get; set; }

public int[] ProductionCountry { get; set; }

public List<string> GenreString { get; set; }

public List<string> KeywordString { get; set; }

public List<string> LanguageString { get; set; }

public List<string> DirectorString { get; set; }

public List<string> CountryString { get; set; }

}

A tanítóhalmaz feltöltését a CreateLearningData metódus végzi, és a bemutatott LearningMovie típus használatával létrehozott learningMovies listában kerülnek ezek a filmek eltárolásra. A tanítóhalmaz kialakítása véletlenszerűen történik a movies listából, méghozzá olyan formában, hogy mindig 30 elemű, viszont a kezdő és végpontja változik (figyelembe véve azt, hogy ne fusson ki a movies lista tartományából), ezért minden egyes futtatásnál más és más filmek kerülnek bele ebbe a tartományba. Ezután for ciklussal végigmegyünk a generált 30 elemes tartományon, és 30 darab LearningMovies típusú filmet hozunk létre, amelyekhez a fentebb tárgyalt adattagokat a movies listából, illetve a tömbök esetén properties-ből adjuk meg.

/// <summary>

/// A véletlenszerű tanítóhalmaz kialakítása

/// </summary>

public void CreateLearningData()

{

Random rndm = new Random();

int startIndex = rndm.Next(0, movies.Count - 30);

int endIndex = startIndex + 30;

for (int i = startIndex; i < endIndex; i++)

{

LearningMovie newMovie = new LearningMovie();

newMovie.Released = movies[i].Released;

newMovie.Runtime = movies[i].Runtime;

newMovie.GenderOfProtagonist = movies[i].GenderOfProtagonist;

newMovie.Genre = properties.GenreContains[i];

newMovie.GenreString = movies[i].GenreString;

FillupFeaturesForCheckbox(featuresForCheckbox.Genre, movies[i].GenreString);

.

.

.

newMovie.TmdbScore = movies[i].TmdbScore;

newMovie.IsPopular = movies[i].IsPopular;

newMovie.Blockbuster = movies[i].IsBlockbuster;

learningMovies.Add(newMovie);

}

}

A tanítóhalmaz feltöltése során feltöltésre kerül a featuresForCheckbox is, amely a FeaturesForCheckbox osztály példányosítása. Az osztály string típusú lista adattagokat tartalmaz azokhoz a kérdésekhez, aminél a tanítóhalmaz filmjeinek tulajdonságaiból választhat a felhasználó az adott kérdésre. A különböző adattagokat a FillupFeaturesForCheckbox metódus tölti fel a tanítóhalmaz kialakítását végző metódusban, mivel mikor a tanítóhalmaz aktuális eleméhez az összetett jellemzők adatait szövegesen is hozzárendeljük, akkor ezeket paraméterként a FillupFeaturesForCheckbox-nak is megadjuk, amely a featuresForCheckbox megfelelő adattagjait feltölti velük, de csak azon elemeivel, amelyek még nem szerepelnek a listában.

/// <summary>

/// Adattagok, amelyekben filmeknek a válaszlehetőségként szolgáló adatai tárolódnak

/// </summary>

internal class FeaturesForCheckbox

{

public List<string> MainActor { get; set; }

public List<string> Genre { get; set; }

public List<string> Keyword { get; set; }

public List<string> Language { get; set; }

public List<string> Director { get; set; }

public List<string> Country { get; set; }

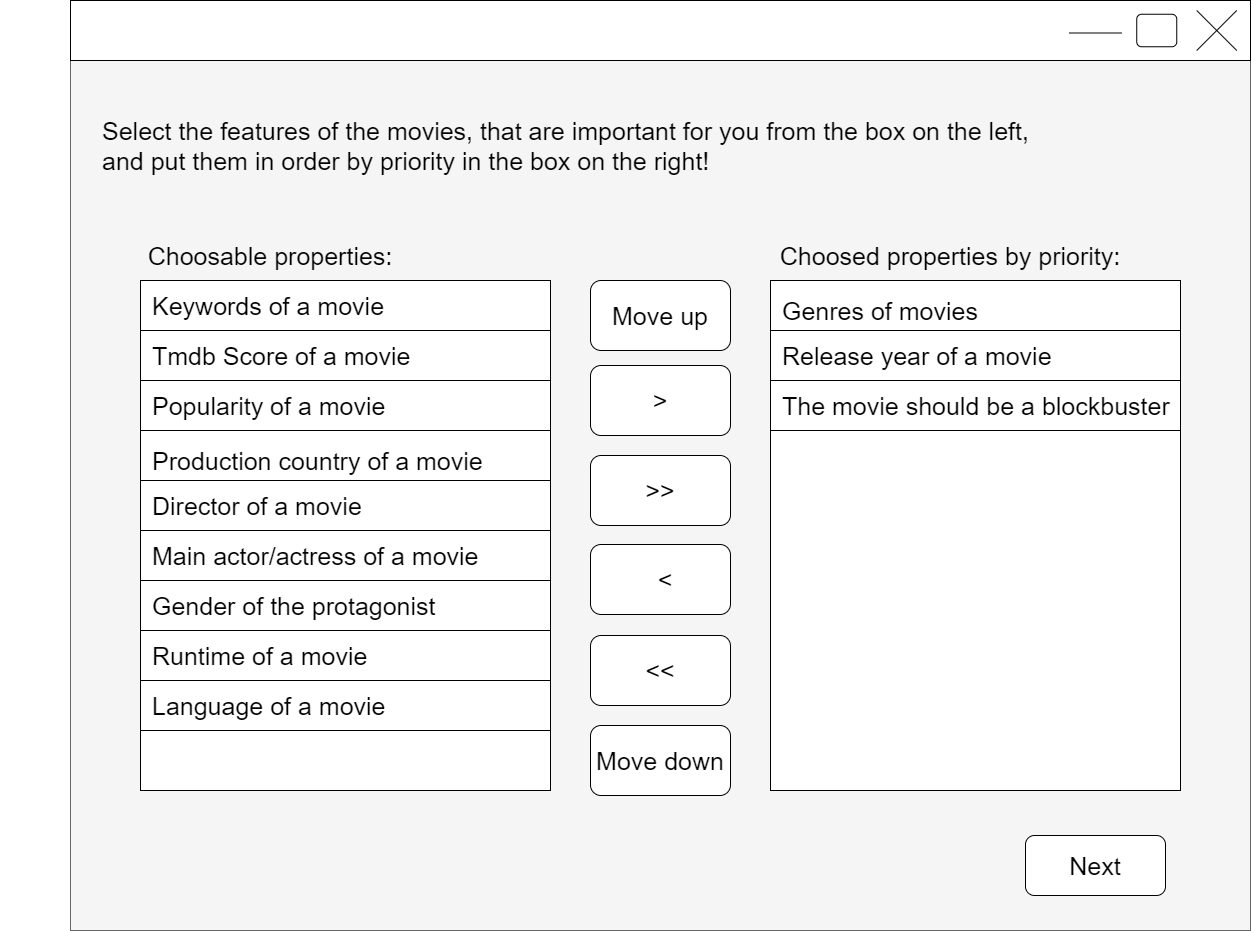
}

A programnak ezen a pontján a kérdések ugyan már megjeleníthetőek a prioritási lista tartalmának felhasználásával, és a válaszlehetőségek kiválasztásához szolgáló checkedlistbox feltöltéséhez szükséges listák is tartalmazzák már a filmek jellemzőit, de a megjelenített kérdéseket még össze kell kötni a megjelenítendő válaszlehetőségekkel.

Mint korábban említésre került, a kérdéseknek a rájuk való válaszolás szempontjából 3 típusa van. Azon kérdéseknél, ahol a felhasználó a kedvenc dolgait választja ki a listából checkboxokkal, illetve az értéket/tartományt beállító csúszkák esetében is minden egyes kérdéshez tartozik egy algoritmus, mivel ezeknél szükség van arra, hogy a válaszlehetőségeket a kérdésekhez igazítsuk. Az igaz/hamis kérdéseknél minden esetben ugyanazt a metódust, a QuestionTrueOrFalse-t hívjuk meg, hiszen itt a válaszlehetőségeknek nem kell igazodni a kérdésekhez.

Ezeket a különböző metódusokat mindhárom kérdéstípus esetén a LoadAnswers metódus hívja meg egy switch case szerkezettel, amely a currentIndex aktuális értékét használja fel indexként a prioritási listában, amit pedig csak a következő kérdésre lépéskor növelünk.

A Genre, Released és Blockbuster 3 különböző típusú jellemző a válaszolás szempontjából, így a következőkben vegyük azt a példát, hogy a felhasználó ezt a 3 jellemzőt választotta ki a korábban bemutatott prioritásválasztási képernyőn a felsorolt prioritási sorrendben. Ez esetben a korábban már bemutatott prioritásválasztó képernyő a következő képpen fog kinézni:



A Next gomb megnyomására Setpriority osztályból áttérünk a QuestionsControl osztályba, ahol az ablak megnyitásakor rögtön lefutó QuestionsControl\_Load metódus fog lefutni először. Ebben a metódusban legelőször is a filmek pontszámait lenullázzuk a SetScore metódussal az eredeti movies listában is, illetve a learningMovies listában is, hogy ha esetleg a program futásakor egymás után többször is lefuttatnánk a programot, akkor ne zavarjanak be a becslésbe az előző futáskor kialakított pontszámok. Ebben a metódusban hívjuk meg az eddig bemutatott metódusokat, amelyek a kérdések és válaszlehetőségek megjelenítésének előkészítéséhez voltak szükségesek, illetve betöltjük az első kérdést, amely ez esetben a Genres jellemzőhöz kapcsolódó kérdés lesz, és a hozzá tartozó válaszlehetőségek.

private void QuestionsControl\_Load(object sender, EventArgs e)

{

SetScore();

InitFeaturesForCheckbox();

CreateLearningData();

CreateQuestions();

LoadQuestion();

LoadAnswers(receivedPriorityList[currentIndex].Id);

if (receivedPriorityList.Count == 1)

{

buttonNext.Visible = false;

buttonFinish.Visible = true;

}

}

A Genres esetében a LoadAnswers a következő metódust fogja meghívni, hiszen a LoadAnswers-ben a switch case szerkezetben a Genres eset mellé ez a metódus lett beállítva.

/// <summary>

/// A felhasználó kedvenc dolgainak kiválasztása kérdéstípus esetében a műfajokhoz kapcsolódó válaszlehetőségek betöltése

/// </summary>

public void GenreQuestion()

{

checkedListBox.Items.Clear();

for (int i = 0; i < featuresForCheckbox.Genre.Count; i++)

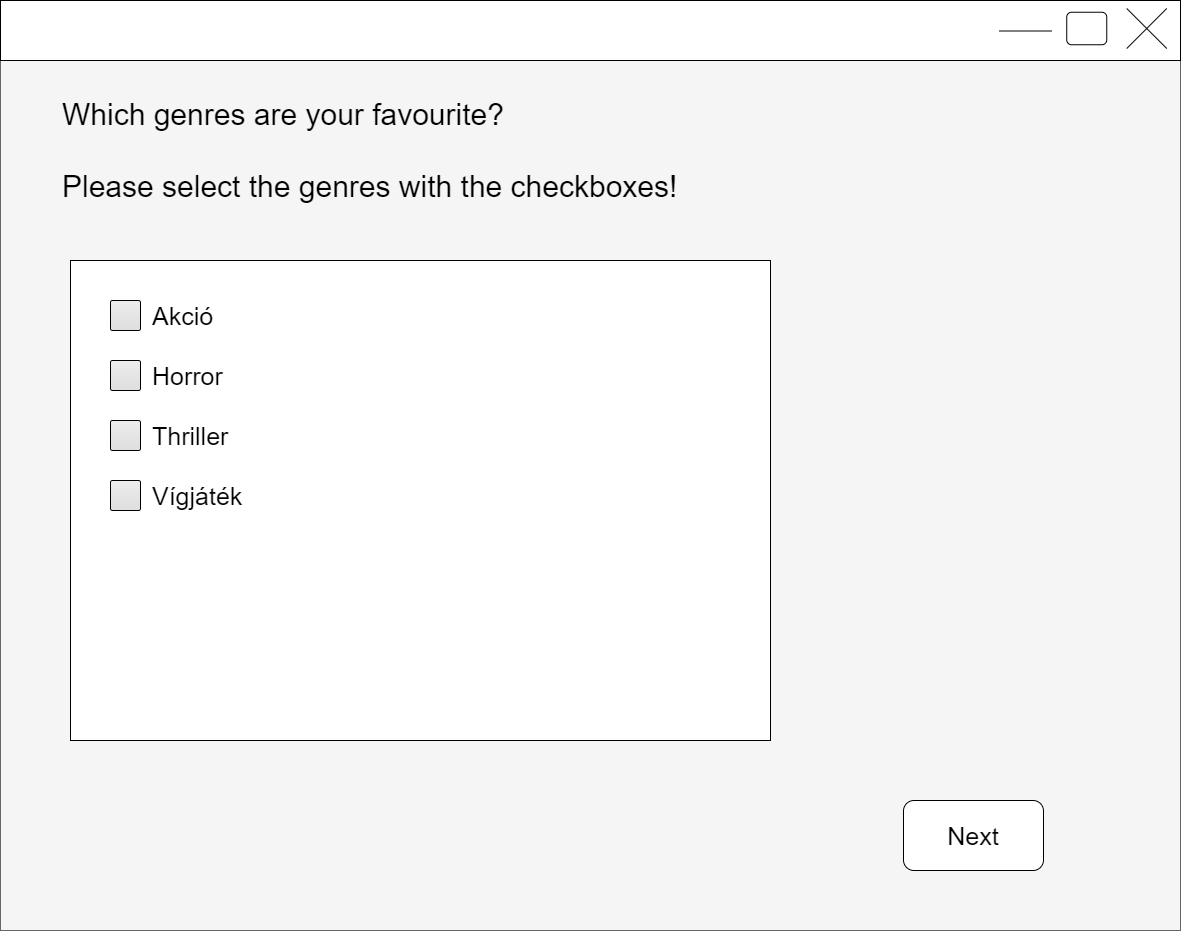
{

checkedListBox.Items.Add(featuresForCheckbox.Genre[i]);

}

}

A QuestionsControl\_Load lefutása után a következő képpen fog kinézni a képernyő:



A felhasználó a kedvenc műfajainak kiválasztása után megnyomja a Next gombot a következő kérdésre lépéshez. Ekkor a Next gomb megnyomásakor lefutó metódus fut le:

/// <summary>

/// A Next gomb megnyomásakor lefutó metódus

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void buttonNext\_Click(object sender, EventArgs e)

{

if (currentIndex < receivedPriorityList.Count - 1)

{

UpdateScore();

currentIndex++;

if (trackBarMin.Visible == true || trackBarMax.Visible==true)

{

trackBarMin.Visible = false;

trackBarMax.Visible = false;

textBoxTrackbarMin.Visible = false;

textBoxTrackbarMax.Visible = false;

checkedListBox.Visible = true;

labelSliderMinMin.Visible = false;

labelSliderMinMax.Visible = false;

labelSliderMaxMin.Visible = false;

labelSliderMaxMax.Visible = false;

}

if (textBoxSearch.Visible == true)

{

textBoxSearch.Visible = false;

labelSearch.Visible = false;

buttonClear.Visible = false;

}

if (trueOrFalse)

{

trueOrFalse = false;

}

LoadQuestion();

LoadAnswers(receivedPriorityList[currentIndex].Id);

if (currentIndex == receivedPriorityList.Count - 1)

{

buttonNext.Visible = false;

buttonFinish.Visible = true;

}

}

}

A metódus először megvizsgálja a QuestionsControl\_Load-hoz hasonlóan, hogy biztosan nem-e az utolsó kérdésről van szó. Ha igen, akkor a Next gomb eltűnik, és megjelenik helyette a Finish. Ellenben ha nem az utolsó, akkor meghívja az UpdateScore metódust, amely a pontszámok kialakításának az alapját adja. A lefutása után megnöveli a currentIndex értékét eggyel, és leellenőrzni, hogy láthatóak-e a csúszkák (trackbarok). Mivel a checkboxos kérdések vannak többségben, így a ritkábbik eset az, hogy a csúszkákat használjuk, ezért az alapértelmezett az, hogy nem láthatóak, csak az érintett kérdésekhez tartozó metódusokban tesszük őket láthatóvá és minden alkalommal eltűntetjük őket és a hozzájuk tartozó elemeket, ha a következő kérdésre ugrunk.

A Genre jellemzőhöz hasonló, checkboxos, több választ is választható kérdések közül a Keyword volt az egyetlen, amely egy keresőt is kapott, mivel a válaszopciók olyan hosszú listát alkottak már 30 film esetén is, hogy kereső volt szükséges hozzá a gyorsabb kereshetőség érdekében.

Hasonló okok miatt, mint a csúszkákat, a keresőt és hozzá tartozó grafikus elemek láthatóságát is le kell ellenőrizni minden kérdésre való továbblépéskor, és eltűntetni, amennyiben láthatóak.

A megkülönböztetés miatt az igaz/hamis kérdések kezeléséhez is bevezetésre került egy bool típusú, osztály szintű változó, amelyet az érintett kérdések metódusánál igazra változtatunk, a következő lépésre tovább lépéskor pedig hamisra.

/// <summary>

/// A pontszámok frissítését végző metódus, amely a Next gomb megnyomására fut le

/// </summary>

public void UpdateScore()

{

if (receivedPriorityList[currentIndex].Id == (int)Features.GenderOfProtagonist)

{

string temp = "";

if (checkedListBox.CheckedItems[0].ToString().Contains("True"))

{

temp = "1";

}

else

{

temp = "2";

}

selectedFromCheckbox.Add(temp);

}

else

{

for (int i = 0; i < checkedListBox.CheckedItems.Count; i++)

{

selectedFromCheckbox.Add(checkedListBox.CheckedItems[i].ToString());

}

}

double score = (receivedPriorityList.Count - receivedPriorityList[currentIndex].Priority) \* 0.1;

switch (receivedPriorityList[currentIndex].Id)

{

case (int)Features.Genre:

ModifyScore(learningMovies => learningMovies.GenreString, score);

break;

case (int)Features.Language:

ModifyScore(learningMovies => learningMovies.LanguageString, score);

break;

case (int)Features.Blockbuster:

ModifyScoreForTrueOrFalse(learningMovies => learningMovies.Blockbuster, score);

break;

.

.

.

}

A metódus elején megvizsgáljuk, hogy esetleg nem-e a GenderOfProtagonist jellemzőről van szó, mivel ott 1 vagy 2 értékekre kell váltanunk az igaz/hamis értékeket, hiszen 1-esekkel és 2-esekkel vannak ezek az adatok eltárolva az adatbázisban. Miután az átalakítás megtörtént a többi igaz/hamis válaszos kérdéssel megegyezően tudunk eljárni.

Ha nem a GenderOfProtagonist jellemzőről van szó, akkor a string típusú osztály szintű selectedFromCheckbox listában eltároljuk a checkboxokkal kiválasztott elemeket. Ezután kiszámítjuk a pontszámot, amelyekkel aktuálisan meg fogjuk növelni azoknak a filmeknek a pontszámát, amelyekre igazak jelen esetben a kiválasztott műfajok. A pontszám kialakítása súlyozással történik a prioritási lista elemeinek prioritási sorrendje segítségével. Ehhez a következő képletet használtam: (a-b)\*0,1, ahol "a" a prioritási lista elemszáma, tehát gyakorlatilag hogy hány db kérdést teszünk fel a felhasználónak, "b" pedig az aktuális kérdés prioritási értéke. (a prioritási listában a legfelső elem, tehát a legfontosabb elem prioritási értéke 0, tehát ez esetben lesz a legmagasabb az a pontszám, amivel növeljük a feltételt teljesítő filmek pontszámát.)

A pontszámok kialakítása után egy switch case szerkezet segítségével meghívjuk a ModifyScore metódusok különböző verziói közül a megfelelőt a kérdéstípustól függően, és paraméterként átadjuk neki a learningMovies listát, illetve a listának a filmek azon jellemzőjéhez tartozó adattagját, amelyben keressük a megadott válaszokkal való egyezéseket, továbbá a kiszámított pontszámot.

A ModifyScore különböző verziói a következőek:

* ModifyScore: az összetett jellemzőkhöz, amik több értéket is felvehetnek, mint pl. a Keyword, Genre
* ModifyScoreSingle: az olyan jellemzőkhöz, amely minden film esetén csak 1 szöveges értéket vehet fel, pl. a film főszereplője
* ModifyScoreForTrueOrFalse: az igaz/hamis értéket felvehető jellemzőkhöz, pl. Blockbuster, GenderOfProtagonist
* ModifíScoreForTrackbar: az olyan jellemzőkhöz, amelyekhez kapcsolódó kérdésekre csúszkával kell válaszolni

A metódusok között csak a bemeneti adatokban, és ezáltal azok kezelésében van eltérés, ettől eltekintve mind a négy metódus logikája ugyanaz: A megadott válaszokat összehasonlítja az összes film aktuálisan vizsgált jellemzőivel, ahol pedig egyezést talál, ott megnöveli az adott film pontszámát az aktuálisan kiszámolttal.

/// <summary>

/// Az adott válaszokkal összehasonlítjuk a filmek adott tulajdonságát, a pontszámukat pedig egyezés esetén megnöveljük

/// </summary>

/// <param name="featureSelector">a learningMovies lista kiválasztott adattagja, amelyben aktuálisan keresni fogunk</param>

/// <param name="score">az UpdateScore metódusban kiszámolt pontszám, amellyel az egyező filmek értékeit növeljük</param>

public void ModifyScore(Func<LearningMovie, List<string>> featureSelector, double score)

{

for (int i = 0; i < selectedFromCheckbox.Count; i++)

{

for (int j = 0; j < learningMovies.Count; j++)

{

if (featureSelector(learningMovies[j]).Contains(selectedFromCheckbox[i]))

{

learningMovies[j].Score += score;

}

}

}

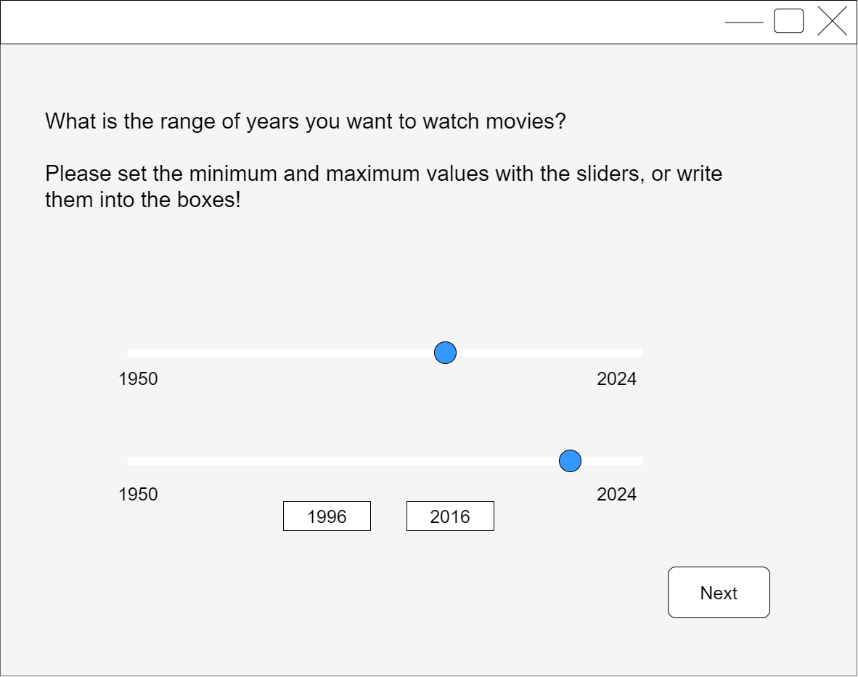
}

A metódus lefutásával az első kérdésre adott válaszok alapján frissítve lettek a filmek pontszámai a tanítóhalmazban. A buttonNext\_Click metódus többi, korábban már bemutatott részének lefutása után továbbléphetünk a következő jellemzőhöz, és egyben kérdéshez, ami esetünkben a Released lesz.

A released jellemzőhöz kapcsolódó kérdés egy olyan típusú kérdés, amelyre csúszkák segítségével kell a felhasználónak válaszolni. 2 darab csúszka használata is szükséges hozzá, hiszen a felhasználó egy tartományt tud beállítani velük.

Az előző kérdésnél a Next gomb megnyomásánál a buttonNext\_Click-ben lefutnak újra a LoadQuestion és a LoadAnswers metódusok, hiszen ezek meghívására minden egyes kérdés esetében szükség van.

Ezáltal a következő képernyő kerül megjelenítésre:



Működését tekintve az értékeket a csúszkákkal tudja beállítani a felhasználó. A csúszka elején és végén címkék jelzik a beállítható minimum és maximum értékeket. A felső csúszka a minimum érték beállításáért felelős, az alsó pedig a maximum értékért. Az alattuk elhelyezett textboxok szinkronizálva vannak a csúszkákkal, tehát ahogyan a felhasználó mozgatja a csúszkát, közben láthatja az aktuális értékét is, mivel a trackbar ValueChanged eseménykezelője frissíti a textbox tartalmát a trackbar tartalmának változásakor. A metódusok ellenőrzik, hogy a minumum értéket beállító trackbar értéke ne lehessen nagyobb mint a maximum értéket beállító értéke, és fordítva.

Amennyiben a felhasználó nem csúszkával szeretné beállítani az értéket, hanem számokkal beírni, akkor az is megvalósítható, ekkor egy enter lenyomása esetén, vagy a textboxból való kikattintás esetében a csúszka is felveszi a textboxban beállított értéket abban az esetben, amennyiben helyesen lett beállítva, különben pedig hibát ír. A beviteli hibák kiküszöböléséről a CheckInputError metódus gondoskodik. Megpróbálja beállítani a trackbar értékét a számmal megadottra, amennyiben nem sikerül neki, akkor jelzi a felhasználónak a beviteli hibát, és automatikusan beállítja a trackbar értékét a másik trackbar értékére.

Mindkét fenti metódus egyaránt kitér arra is, hogy arányosítani kell-e az értékeket (pl. a Tmdb pontszám esetén igen), mert ez esetben a trackbar értékei 100-zal meg vannak szorozva annak érdekében, hogy lehessen lebegőpontos számokat is beállítani, viszont ezért 100-zal osztani kell a valós eredmény eléréséhez, megjelenítéséhez. Erre azért van szükség, mert C#-ban a trackbar csak egész számokat tud kezelni.

A metódus, amelyet a LoadAnswers ez esetben be fog tölteni a következő lesz:

/// <summary>

/// A csúszkás válaszlehetőséggel rendelkező kérdések közül a megjelenés évének esetében meghívandó metódus a válaszokhoz

/// </summary>

public void ReleasedQuestion()

{

SetTrackbarProperties(trackBarMax, textBoxTrackbarMax, 1950, 2024, 2001, false, true,

new int[] { 96, 210 }, new int[] { 317, 286 }, new int[] { 93, 258 }, new int[] { 451, 258 }, labelSliderMaxMin, labelSliderMaxMax);

SetTrackbarProperties(trackBarMin, textBoxTrackbarMin, 1950, 2024, 2000, false, true,

new int[] { 96, 131 }, new int[] { 197, 286 }, new int[] { 93, 179 }, new int[] { 451, 179 }, labelSliderMinMin, labelSliderMinMax);

}

A metódusban a SetTrackbarProperties metódus kerül kétszer is meghívásra, mivel ez esetben két trackbar beállítása szükséges a válaszoláshoz. A metódus, mint a neve is mutatja, a trackbarok tulajdonságainak beállításáért felelős, így különböző paraméterértékekkel hívjuk meg őket az egyes csúszkák esetében. A metódusban a checkboxos kérdésekhez tartozó grafikus elemeket is elrejtjük átmenetileg, és helyüket átveszik a trackbarok. Mind a 3 csúszkás kérdéshez ugyanazokat a trackbarokat használjuk fel, viszont az egyiknél tartományt állítunk be velük, a másiknál csak alsó határértéket, illetve más és más értékek lesznek a különböző határértékek minden kérdés esetében, ezért szükséges a trackbarok jellemzőinek részletes testreszabása minden ilyen típusú metódus, és azon belül trackbar esetén. A metódus elején a checkInputError osztály szintű bool változó értékét hamisra állítjuk, majd a végén visszaállítjuk igazra. Erre azért van szükség, mert a trackbarok tulajdonságainak beállításakor előfordulhat, hogy mikor az egyik értéke már be van állítva, de a másik még nincs, akkor a minimum érték a maximum felé megy, vagy fordítva, és a program hamisan jelez hibaüzenetet a felhasználó felé.

/// <summary>

/// A trackbarok tulajdonságainak beállítását végző metódus

/// </summary>

/// <param name="trackbar">A csúszka, amelynek a jellemzőit módosítjuk</param>

/// <param name="textbox">A csúszkához tartozó textbox</param>

/// <param name="min">A beállítható minimum érték</param>

/// <param name="max">A beállítható maximum érték</param>

/// <param name="startValue">A beállítandó kezdőérték</param>

/// <param name="scaleInput">Annak beállítása, hogy kell-e arányosítani a beállított értékeket</param>

/// <param name="rangeInput">Annak beállítása, hogy tartományt adunk-e meg, vagy csak alsó/felső határértéket</param>

/// <param name="trackbarLocation">A csúszka helyzetének beállítása</param>

/// <param name="textboxLocation">A csúszkához tartozó textbox helyzetének beállítása</param>

/// <param name="labelMinLocation">A beállítható minimum értéket megjelenítő címke helyzete</param>

/// <param name="labelMaxLocation">A beállítható maximum értéket megjelenítő címke helyzete</param>

/// <param name="labelMin">A beállítható minimum értéket megjelenítő címke értéke</param>

/// <param name="labelMax">A beállítható maximum értéket megjelenítő címke értéke</param>

public void SetTrackbarProperties(System.Windows.Forms.TrackBar trackbar, System.Windows.Forms.TextBox textbox,

int min, int max, int startValue, bool scaleInput, bool rangeInput, int[] trackbarLocation, int[] textboxLocation,

int[] labelMinLocation, int[] labelMaxLocation, Label labelMin, Label labelMax)

{

checkValueError=false;

scale = scaleInput;

range= rangeInput;

checkedListBox.Visible = false;

trackbar.Visible = true;

textbox.Visible = true;

labelMin.Visible = true;

labelMax.Visible = true;

trackbar.Minimum = min;

trackbar.Maximum = max;

trackbar.Value = startValue;

checkValueError = true;

trackbar.Location = new Point(trackbarLocation[0], trackbarLocation[1]);

textbox.Location = new Point(textboxLocation[0], textboxLocation[1]);

labelMin.Location=new Point(labelMinLocation[0], labelMinLocation[1]);

labelMax.Location= new Point(labelMaxLocation[0], labelMaxLocation[1]);

if (scale)

{

labelMin.Text = (min / 100).ToString();

labelMax.Text = (max / 100).ToString();

}

else

{

labelMin.Text = min.ToString();

labelMax.Text = max.ToString();

}

}

/// <summary>

/// Az igaz/hamis kérdések esetében meghívandó algoritmus a válaszokhoz

/// </summary>

public void QuestionTrueOrFalse()

{

trueOrFalse = true;

checkedListBox.Items.Clear();

checkedListBox.Items.Add(true);

checkedListBox.Items.Add(false);

}

A csúszkák beállítása után a felhasználó újra a már bemutatott Next gomb megnyomásával tud továbblépni. A buttonNext\_Click tartalma újra lefut, eltérés csak abban van az előző kérdéshez képest, hogy mivel más típusú kérdésről van szó, ezért az UpdateScore metódus ez esetben a ModifyScoreForTrackbar metódust fogja meghívni, mivel a switch case szerkezetben a Released jellemzőhöz ez lett beállítva. A metódus a range változó ellenőrzésével külön veszi azt az esetet mikor tartomány lett megadva, illetve azt is, mikor csak alsó határérték. A Tmdb pontszámnál itt már nem szükséges arányosítani, mivel a feltétel a csúszkához tartozó textbox eredményét veszi át, ott pedig már a megjelenítés pillanatában megtörténik az arányosítás, így már ebben a metódusban is az arányosított értéket használjuk.

public void ModifyScoreForTrackbar(Func<LearningMovie, double> featureSelector, double score)

{

for (int i = 0; i < learningMovies.Count; i++)

{

if (featureSelector(learningMovies[i]) >= float.Parse(textBoxTrackbarMin.Text) && range == false)

{

learningMovies[i].Score += score;

}

else if (featureSelector(learningMovies[i]) >= float.Parse(textBoxTrackbarMin.Text) && featureSelector(learningMovies[i]) <= float.Parse(textBoxTrackbarMax.Text) && range == true)

{

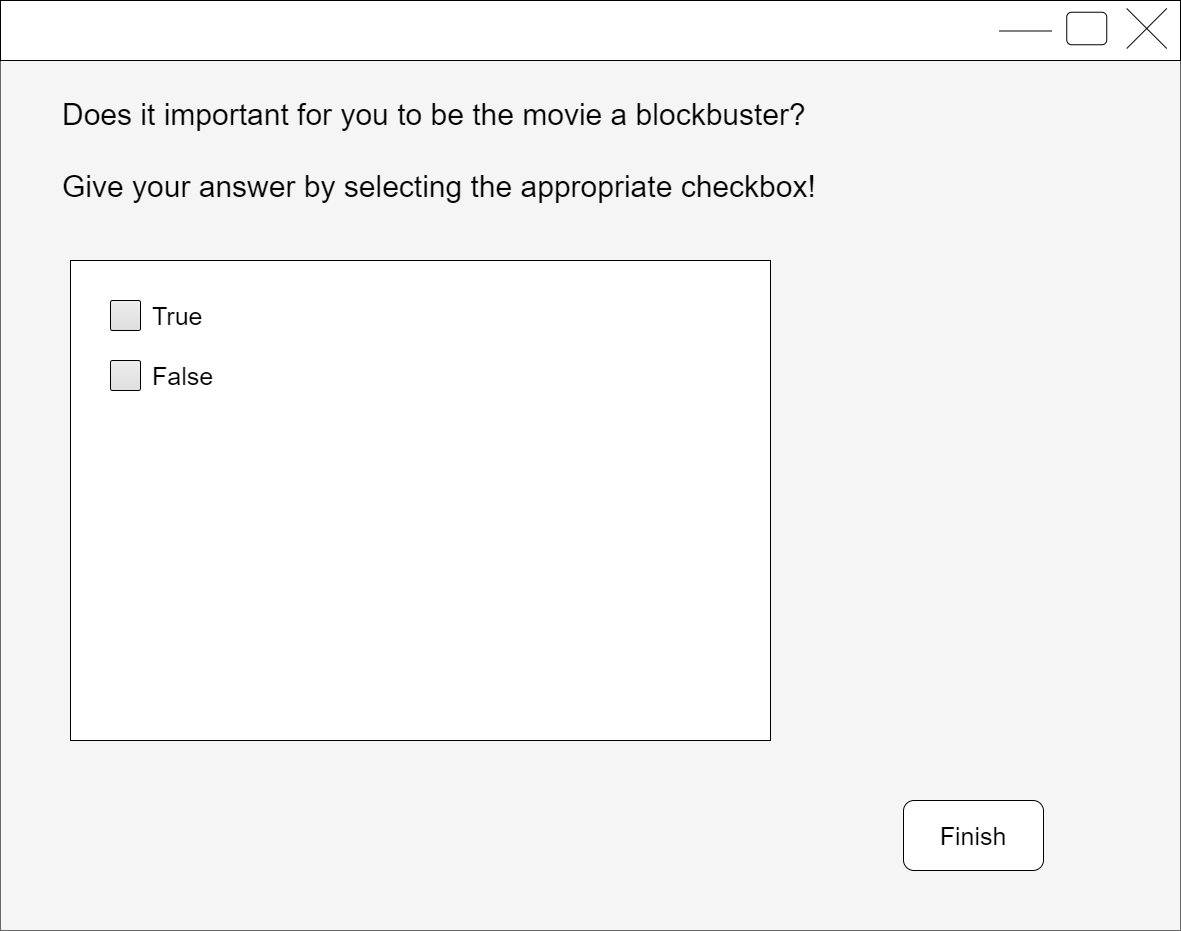
learningMovies[i].Score += score;

}

}

}

A buttonNext\_Click lefutása után eltűnnek a trackbarok és a hozzá kapcsolódó grafikus objektumok, és újra megjelennek a checkboxos kérdésekhez tartozó elemek. A harmadik kérdéstípus az igaz/hamis, ilyen típusú kérdés tartozik a Blockbuster jellemzőhöz is, amely 3. elemnek lett választva a prioritási listában. Ez esetben a meghívandó metódusok megegyeznek a Genre tulajdonságnál már bemutatottakkal annyi különbséggel, hogy a LoadAnswers metódus a QuestionTrueOrFalse-t hívja meg, mint minden igaz/hamis kérdés esetében.



A metódus elején az osztályszintű trueOrFalse bool változót igazra állítjuk. Erre azért van szükség, mert a többi checkboxos, de nem igaz/hamis kérdésnél egyszerre több elemet is kiválaszthatunk válaszként a filmek esetén, viszont igaz/hamis kérdésnél ennek értelemszerűen nincs értelme. Ezért a checkedListBox\_ItemCheck metódussal igaz trueOrFalse érték esetében ha bepipáljuk bármelyik checkboxnak az értékét, akkor az összes többinél ki kell venni a pipát. A checkedlistbox adatforrása ebben az esetben csak 2 elemből fog állni, a true és false-ból.

/// <summary>

/// Az igaz/hamis kérdések esetében meghívandó algoritmus a válaszokhoz

/// </summary>

public void QuestionTrueOrFalse()

{

trueOrFalse = true;

checkedListBox.Items.Clear();

checkedListBox.Items.Add(true);

checkedListBox.Items.Add(false);

}

/// <summary>

/// Annak kezelése, hogy csak egy checkbox lehessen bepipálva igaz/hamis kérdések esetében

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void checkedListBox\_ItemCheck(object sender, ItemCheckEventArgs e)

{

if (trueOrFalse)

{

for (int i = 0; i < checkedListBox.Items.Count; i++)

{

if(i !=e.Index)

{

checkedListBox.SetItemChecked(i, false);

}

}

}

}

A válasz kiválasztása után ezúttal mivel az utolsó kérdésről van szó, ezért a 2. kérdésről való továbblépéskor lefutó buttonNext\_Click metódus eltűntette a Next gombot, helyette pedig megjelent a Finish gomb, amivel a felhasználó továbbléphet az ajánlott filmek kilistázásához.

/// <summary>

/// A finish gomb megnyomásakor lefutó metódus

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void buttonFinish\_Click(object sender, EventArgs e)

{

UpdateScore();

MainDecTree mainDecTree = new MainDecTree();

mainDecTree.BuildTree(learningMovies, movies, properties);

if (FinishButtonClicked != null)

{

FinishButtonClicked(this, EventArgs.Empty);

}

}

A pontszámokat az utolsó kérdés esetében is frissíteni kell, tehát ez esetben is meghívjuk az UpdateScore metódust. A switch case szerkezet ez esetben a ModifyScoreForTrueOrFalse metódust hívja meg, amelynek logikája megegyezik a többi ModifyScore típusú metóduséval.

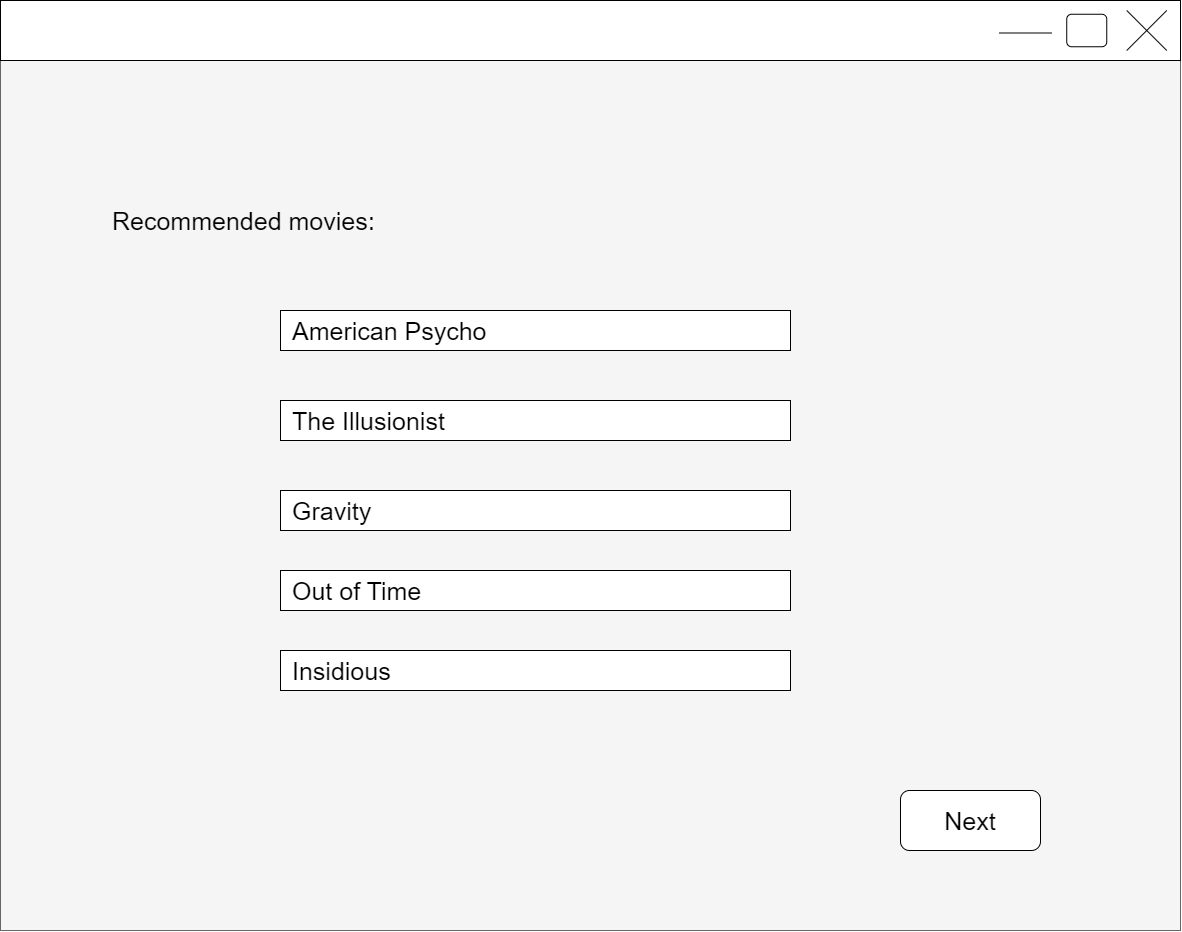
A buttonFinish\_Click metódus többi részében példányosítjuk a fő becslést végző MainDecTree osztályt, és BuildTree metódusának átadjuk a megfelelő paramétereket ahhoz, hogy a tanítóhalmazon kialakított pontszámok alapján megbecsülje az egész adathalmaz pontszámait, illetve a FinishButtonClicked eseménykezelő segítségével továbblépünk a Results UserControl osztályra (hasonlóan, mint a SetPriority Usercontrolról a QuestionsContreol-ra), ahol a becsült eredményeket megjelenítjük.

A fő becslés tehát a MainDecTree osztály Buildtree metódusában található, logikája pedig teljes mértékben megegyezik a DecTreeForTmdb osztályban megvalósított, a filmek Tmdb pontszámára vonatkozó becsléssel.

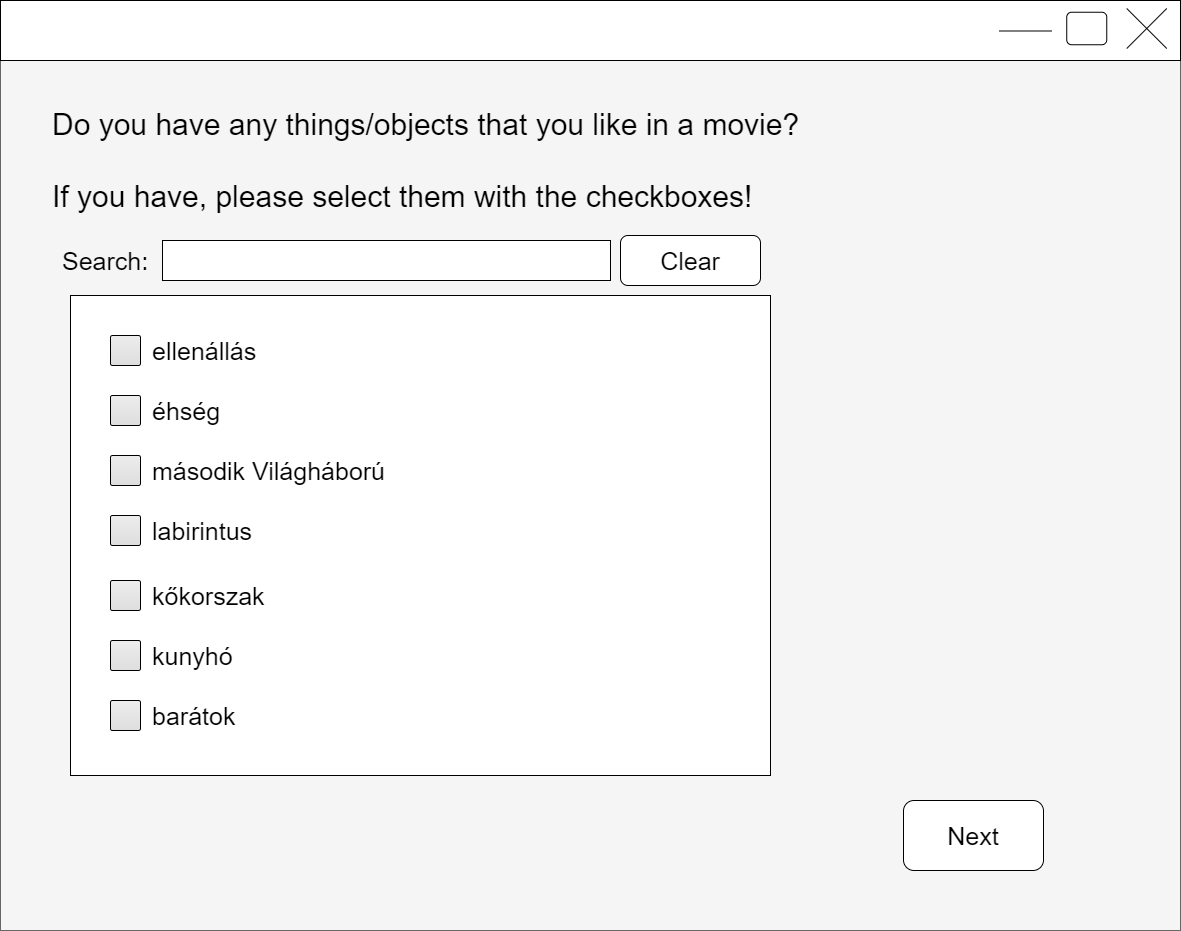
Mindkét becslés esetében regressziós becslésről van szó, hiszen a Tmdb pontszám, és az általunk kialakított pontszámok is folytonos értékek.

A Tmdb pontszámra vonatkozó becslésnél csak a műfajok és a kulcsszavak alapján történt a becslés, a fő becslésnél viszont mind a 12 darab jellemzőt, amire kérdések irányulnak, figyelembe vesszük, és ezek alapján történik a pontszámok megbecslése. A jellemzőkön természetesen itt is először átalakításokat kell végeznünk annak érdekében, hogy a különböző típusú adattagokat az ML.NET által kezelhető formába alakítsuk.

A becslés végén a kapott pontszámokat a movies lista Score adattagjaiban tároljuk el filmenként. A Results osztályban a listát a Score adattagok értékei szerint csökkenő sorrendbe rendezzük, és az első 5 elemét jelenítjük meg végeredményként 5 darab textboxban, ez lesz az 5 legmagasabb pontszámmal rendelkező film a listából, azaz a felhasználó igényeinek a becslés alapján a legjobban megfelelő filmek.



A checkboxos kérdéseknél említésre került, hogy a Keywords jellemző esetében egy kereső megvalósítása is szükséges volt a hosszú listában való gyorsabb, és könnyebb kereshetőség érdekében. A kereső megvalósítása a QuestionsControl FilterCheckedListBox algoritmusában történik.



A kulcsszavak esetében a checkedlistbox tartalma értelemszerűen a tanítóhalmaz összes kigyűjtött kulcsszava lesz, ezeken történik majd a keresés. Az algoritmus akkor fut le, miután a felhasználó beírta a keresőmezőbe az általa keresni kívánt kifejezést, és megnyomja az enter billentyűt. A metódus elején végigmegyünk a checkedlistbox összes elemén, és minden egyes elemre megvizsgáljuk, hogy a selectedFromCheckbox tartalmazza-e az adott értéket. (A metódus legelső lefutásakor, tehát a legelső keresésnél még üres lesz) Ha az adott elem be van pipálva, tehát ki van választva, de a selectedFromCheckbox listának nem eleme, akkor hozzáadjuk, ha pedig nincs kiválasztva az adott elem, akkor pedig eltávolítjuk a listából. Ezzel a logikával a selectedFromCheckbox lista tartalmát minden egyes kereséskor aktualizáljiuk. Ezután a checkedListbox tartalmát töröljük, és a textbox tartalmát összehasonlítjuk a filmekből kigyűjtött kulcsszavakkal. Amelyik tartalmazza a keresendő kifejezést, az a frissített checkedListbox eleme lesz. Végül újból végigmegyünk a frissített checkedListbox elemein, és ha az adott elemet tartalmazza a selectedFromCheckbox lista, akkor a frissített listában is bepipáljuk, tehát kiválasztjuk.

Több keresést is végezhetünk egymás után, viszont az utolsó keresés után bepipált elemek az UpdateScore metódussal kerülnek bele a selectedFromCheckbox listába, mint ahogyan a többi, keresővel nem rendelkező kérdés esetében is történik.

A kereső melletti Clear gombra kattintva törölhetjük a szűrőt, és megjeleníthetjük újra az összes kulcsszót. Ebben az esetben is a FilterCheckedListBox metódust hívjuk meg, csak üres stringgel (””), amelyet minden elem tartalmaz, ezért mindegyik megjelenítésre kerül.

/// <summary>

/// Kereső a Keywords jellemzőhöz kapcsolódó kérdéshez

/// </summary>

/// <param name="searchText">A keresett kifejezés</param>

private void FilterCheckedListBox(string searchText)

{

string itemText;

for (int i = 0; i < checkedListBox.Items.Count; i++)

{

itemText = checkedListBox.Items[i].ToString();

if (checkedListBox.GetItemChecked(i)==true)

{

if (!selectedFromCheckbox.Contains(itemText))

{

selectedFromCheckbox.Add(itemText);

}

}

else

{

selectedFromCheckbox.Remove(itemText);

}

}

checkedListBox.Items.Clear();

for(int i = 0; i < featuresForCheckbox.Keyword.Count; i++)

{

if (featuresForCheckbox.Keyword[i].ToLower().Contains(searchText))

{

checkedListBox.Items.Add(featuresForCheckbox.Keyword[i]);

}

}

for (int i = 0; i < checkedListBox.Items.Count; i++)

{

itemText = checkedListBox.Items[i].ToString();

if (selectedFromCheckbox.Contains(itemText))

{

checkedListBox.SetItemChecked(i, true);

}

}

}

1. https://www.sqlite.org/index.html [↑](#footnote-ref-1)
2. https://www.sqlite.org/whentouse.html [↑](#footnote-ref-2)
3. https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database [↑](#footnote-ref-3)
4. https://system.data.sqlite.org/index.html/doc/trunk/www/index.wiki [↑](#footnote-ref-4)
5. https://cloud.google.com/learn/artificial-intelligence-vs-machine-learning [↑](#footnote-ref-5)
6. https://en.wikipedia.org/wiki/ML.NET [↑](#footnote-ref-6)
7. https://learn.microsoft.com/hu-hu/dotnet/machine-learning/how-does-mldotnet-work [↑](#footnote-ref-7)
8. https://dotnet.microsoft.com/en-us/learn/ml-dotnet/what-is-mldotnet [↑](#footnote-ref-8)
9. https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/transforms?WT.mc\_id=dotnet-35129-website [↑](#footnote-ref-9)
10. https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/tasks?WT.mc\_id=dotnet-35129-website [↑](#footnote-ref-10)
11. https://dl.acm.org/doi/abs/10.1145/358916.358995 [↑](#footnote-ref-11)
12. https://www.sciencedirect.com/science/article/abs/pii/S0950705106000189 [↑](#footnote-ref-12)
13. https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/metrics?WT.mc\_id=dotnet-35129-website [↑](#footnote-ref-13)
14. https://vitalflux.com/micro-average-macro-average-scoring-metrics-multi-class-classification-python/#What\_Why\_of\_Micro\_Macro-averaging\_and\_Weighting\_metrics [↑](#footnote-ref-14)
15. https://learn.microsoft.com/en-us/dotnet/machine-learning/how-to-guides/save-load-machine-learning-models-ml-net?WT.mc\_id=dotnet-35129-website [↑](#footnote-ref-15)