

**MISKOLCI EGYETEM**  
Gépészmérnöki és Informatikai Kar  
Matematikai Intézet  
Analízis Intézeti Tanszék

**Sorszám: GEMAN/PJ8HD2/2024/BSc**

## **SZAKDOLGOZATI FELADAT**

**Siska Dávid**

gazdaságinformatikus jelölt  
részére

**A szakdolgozat tárgyköre:** Szoftverfejlesztés

**A szakdolgozat címe:**

**Kérdőív alapú adaptív ajánlórendszer fejlesztése**

**A feladat részletezése:**

A dolgozat célja egy olyan ajánlórendszer elkészítése, amely dinamikusan létrehozott kérdőívek segítségével igyekszik segíteni a felhasználót egy nagy elemszámú adathalmazból a megfelelő elem kiválasztásában. (Ilyen lehet például, hogy ha egy megfelelő terméket keres egy katalógusban.) A dolgozatban röviden át kell tekinteni a probléma hagyományos megoldási módjait (például kulcsszó alapú keresők és szűrők alkalmazását), majd részletesen be kell mutatni, hogy hogyan lehet a felhasználói visszajelzésekre alapozva, automatikusan generálni olyan kérdőívet, amely segítségével a lehető legkevesebb lépésben, a lehető legnagyobb megbízhatósággal megtalálható a keresett elem (a korábban felvitt visszajelzések alapján).

**Témavezető(k):** Dr. Glavosits Tamás, egyetemi docens

**Konzulens(ek):**

**A feladat kiadásának ideje:** 2024. 03. 01.

.....  
Dr. Szigeti Jenő  
szakfelelős

1.

szükséges.

A szakdolgozat feladat módosítása

nem szükséges.

.....  
dátum

.....  
témavezető(k)

2. A feladat kidolgozását ellenőriztem:

témavezető (dátum, aláírás):

konzulens (dátum, aláírás):

.....  
.....  
.....

.....  
.....  
.....

3. A szakdolgozat beadható:

.....  
dátum

.....  
témavezető(k)

4. A szakdolgozat ..... szövegoldalt

..... program protokollt (listát, felhasználói leírást)

..... elektronikus adathordozót (részletezve)

.....

..... egyéb mellékletet (részletezve)

.....

tartalmaz.

.....  
dátum

.....  
témavezető(k)

5.

bocsátható.

A szakdolgozat bírálatra

nem bocsátható.

A bíráló neve: .....

.....  
dátum

.....  
szakfelelős

6. A szakdolgozat osztályzata

a témavezető javaslata: .....

a bíráló javaslata: .....

a szakdolgozat végleges eredménye: .....

Miskolc, .....

.....  
a Záróvizsga Bizottság Elnöke

## EREDETISÉGI NYILATKOZAT

Alulírott Siska Dávid; Neptun-kód: PJ8HD2 a Miskolci Egyetem Gépészmérnöki és Informatikai Karának végzős, gazdaságinformatika szakos hallgatója ezennel büntetőjogi és fegyelmi felelősségem tudatában nyilatkozom és aláírással igazolom, hogy „Kérdőív alapú adaptív ajánlórendszer fejlesztése” című szakdolgozatom saját, önálló munkám; az abban hivatkozott szakirodalom felhasználása a forráskezelés szabályai szerint történt.

Tudomásul veszem, hogy szakdolgozat esetén plágiumnak számít:

- szószerinti idézet közlése idézőjel és hivatkozás megjelölése nélkül;
- tartalmi idézet hivatkozás megjelölése nélkül;
- más publikált gondolatainak saját gondolatként való feltüntetése.

Alulírott kijelentem, hogy a plágium fogalmát megismertem, és tudomásul veszem, hogy plágium esetén szakdolgozatom visszautasításra kerül.

Miskolc, .....év .....hó .....nap

.....  
Hallgató



## **SZAKDOLGOZAT**

# **KÉRDŐÍV ALAPÚ ADAPTÍV AJÁNLÓRENDSZER FEJLESZTÉSE**

Készítette:

Siska Dávid

Neptunkód: PJ8HD2

Gazdaságinformatikus

Témavezető: Dr. Glavosits Tamás, egyetemi docens

**Miskolci Egyetem**

**2024**

# Tartalomjegyzék

1. Bevezetés .....	1
2. Ajánlórendszer építéséhez szükséges komponensek bemutatása .....	4
2.1 Alternatív keresők bemutatása .....	4
2.1.1 Kulcsszó alapú keresők .....	4
2.1.2 Szemantikus keresők .....	4
2.1.3 NLP (Természetes nyelvfeldolgozás) .....	5
2.1.4 Faceted kereső .....	5
2.1.5 Kép alapú keresők (CBIR) .....	6
2.2 SQLite bemutatása .....	6
2.3 Az adatbázis felépítése .....	7
2.3.1 Az adatbázis szerkezete .....	7
2.3.2 Az adatbázis ER modellje .....	11
2.3.3 Az adatbázis relációs modellje .....	13
2.4 Kézzel készített döntési fa a film nyelvére vonatkozóan .....	14
2.5 Kézzel készített döntési fa hatékonysága .....	16
2.6 Folyamatábra az adatbázis adatainak feldolgozásáról .....	17
2.7 ML.NET bemutatása .....	18
2.7.1 A mesterséges intelligencia és a gépi tanulás kapcsolata .....	18
2.7.2 Mi az az ML.NET? .....	19
2.8 Hogyan néz ki egy ML.NET alkalmazás? .....	19
2.8.1 MLContext létrehozása .....	19
2.8.2 Adatok betöltése .....	20
2.8.3 Adatok átalakítása .....	20
2.8.4 Algoritmus kiválasztása .....	21
2.8.5 Modell tanítása .....	23
2.8.6 Modell kiértékelése .....	23
2.8.7 A modell kimentése, használata .....	25
2.9 Algoritmusok hatékonyságának összehasonlítása .....	26
2.10 Gyakorlás céljából készített becslő algoritmusok hatékonyságának összehasonlítása .....	29
3. Az implementált program ajánlórendszerének felépítése .....	30
3.1 SetPriority osztály .....	30
3.2 QuestionsControl osztály .....	33

3.2.1 Kérdések kialakítása .....	34
3.2.2 Tanítóhalmaz kialakítása, feltöltése .....	35
3.2.3 A kérdések összekapcsolása a jellemzőkkel, és a válaszokkal .....	37
3.2.4 A checkboxokat használó, és több válaszopciót is elfogadó kérdések kezelése .....	38
3.2.5 A csúszkákat használó kérdések kezelése .....	43
3.2.6 Az igaz/hamis kérdések kezelése .....	47
3.2.7 Az eredmények becslése .....	49
3.2.8 Kereső megvalósítása a kulcsszavakhoz .....	50
4. A program használatának a bemutatása .....	53
4.1 A program funkciói .....	53
4.1.1 Az adatbázis tartalmának megjelenítése .....	53
4.1.2 Az adatbázis cseréje .....	54
4.1.3 Filmek kezelése .....	55
4.1.4 Film ajánlása .....	56
4.2 Futtatható kód előállítása a forráskódból .....	59
4.2.1 Fordítás Debug módban .....	59
4.2.2 Fordítás Release módban .....	60
4.2.3 Debug és Release mód összehasonlítása .....	60
5. Összefoglalás .....	61
6. Summary .....	62
7. Köszönetnyilvánítás .....	63
8. Irodalomjegyzék .....	63

## Melléklet

A. A TableInserts program felépítése .....	1
A.1 SqlConnection osztály .....	2
A.2 Program osztály .....	3
A.3 Table osztály .....	3
A.3.1 Táblát feltöltő metódusok .....	3
A.3.2 Kapcsolótáblát feltöltő metódusok .....	5
B. MovieRecommendationSystem program felépítése .....	6
B.1 Movie osztály .....	7
B.2 A MovieRecommendationSystem és a TableInserts program összekapcsolása .....	7
B.3 PropertiesForDecTree osztály .....	8
B.4 SqlConnection osztály .....	8

B.5 Algorithms osztály .....	10
B.6 HandmadeLanguageDecTree osztály .....	13
B.7 ProgressBar osztály .....	14
B.8 Main osztály .....	14
B.8.1 Table gomb.....	15
B.8.2 Change Database gomb .....	16
B.8.3 Manage Movie gomb .....	16
B.9 Becslésekre vonatkozó osztályok.....	21
B.9.1 Tmdb osztály .....	21
B.9.2 TmdbPredict osztály.....	21
B.9.3 DecTreeForTmdb.....	22
B.10 MeasureAccuracy osztály.....	24

# 1. Bevezetés

Napjainkban az embereknek számos döntést kell meghozniuk nap mint nap, melyek sokszor rendkívül fontosak, amelyek nagy átgondoltságot, hosszas gondolkodást igényelhetnek, hiszen hosszan tartó, meghatározó következményeik lehetnek. Máskor kisebb hangsúlyú, kisebb jelentőséggel bíró döntéseket kell meghozni, mint például egy-egy termék megvásárlása, viszont ezek is az emberek feladatai közé tartoznak, és nehéz dolguk lehet a választásban.

Bármilyen döntésről is legyen szó, az emberek mindig próbálnak mérlegelni, figyelembe venni az egyes döntéseknek a lehetséges következményeit, tehát észérvek alapján meghozni egy-egy választást, viszont ez sokszor nem egyszerű, és nem is sikerül. Az emberek sokszor nem rendelkeznek kellő információval és háttértudással egy-egy tématerületet illetően ahhoz, hogy ezeket a tématerületeket érintő kérdésekben logikusan tudjanak dönteni. Ilyenkor a kellő háttértudás hiánya miatt az érzelmeik, „megérzéseik” a mérvadóak egy-egy döntés meghozatala során, és ezáltal előfordulhat, hogy nem a legjobb, legcélszerűbb döntést hozzák meg a lehetséges opciók közül.

A gépek, programok alapvetően nem rendelkeznek semmilyen témát illetően semmilyen tudással. Viszont ha az adott tématerületet illetően a logikus döntéshozáshoz szükséges adatokat megkapják, és emellé párosul egy olyan algoritmus, amely a kellő alapossággal és módszerekkel képes elemezni, feldolgozni ezeket az adatokat, akkor egy olyan mögöttes összefüggéseket figyelembe vevő megközelítés alapján tudnak döntést hozni, amelyet a felhasználók a felszíni tulajdonságok vizsgálatával észre sem vennének. Ezt szokás gépi tanulásnak nevezni [1].

A gépi tanulás egy altípusa a mélytanulás, amelyet pedig reprezentációs tanúláshoz használnak [2]. Ennek célja, hogy nagy mennyiségű jelöletlen adatból, amelyek lehetnek képek, hangok, szövegek, olyan őket reprezentáló beágyazási vektorokat hozzanak létre, amelyek nem a felszíni tulajdonságaik alapján írják le az egyedeket, hanem a mélyebb, mögöttes tartalmi összefüggésekre helyezik a hangsúlyt [1].

Az ajánlórendszer kifejezést olyan eszközök, szoftverek összességére használjuk, amelyek értékes információkat tudnak szolgáltatni a felhasználók számára abban a formában, hogy hétköznapi kérdésekre próbálnak választ adni egy-egy döntés formájában. Ezt olyan módon teszik, hogy a felhasználó igényeinek megfelelő dolgokat, termékeket ajánlanak neki. A mai világban, ahol egyre több-és több termék elérhető és rendelhető meg az interneten, ez egy rendkívül hasznos funkció tud lenni a felhasználók, vásárlók számára, hiszen egyébként töménytelen mennyiségű választási lehetőségből kellene válogatniuk és választaniuk, ami megbonyolítja az egész döntési folyamatot. A legtöbbször ezeket az ajánlórendszereket a felhasználók nem is tudatosan használják, hiszen az általuk használt weboldalak, webáruházak, boltok online oldalai is beépítve, mivel a boltoknak is érdekük, hogy a vásárlók minél hamarabb megtalálják a számukra minél tökéletesebb termékeket [3]. Az ajánlórendszereket gyakran ötvözik



mélytanulási módszerekkel is annak érdekében, hogy növeljék hatékonyságukat, és különböző felmerülő problémákat küszöböljenek ki [2].

Az ajánlórendszerek működési koncepciójából adódik, hogy minden ajánlórendszernek szüksége van bemeneti adatokra, amelyekkel a felhasználó igényeinek megfelelő javaslatokat tud tenni. A bemeneti adatok a következők lehetnek [3]:

- Adatok, amelyeket a felhasználó jelenléte hoz be a rendszerbe. A felhasználó személyes adatai, mint pl. a neve, életkora, érdeklődési köre, munkaköre stb.
- Az egyedekre vonatkozó jellemzők, amelyek nagyban függenek az egyed jellegétől.
- A felhasználótól kapott visszajelzések, amelyek lehetnek az ajánlórendszerbe tudatosan bevitt adatok, mint pl. értékelések, vagy akár korábbi vásárlások, amelyeket fel lehet használni a jövőben az ajánlásokhoz.

Az ajánlórendszerek alapját az osztályozók adják, ezért fontos, hogy a különböző osztályozók közül melyiket választjuk, hiszen nagyban tudja befolyásolni a rendszer pontosságát. Osztályozáshoz választhatjuk például a K-Közép, Naiv Bayes, az SVM (vektorgépek támogatása) osztályozókat, vagy akár a döntési fákat is. A döntési fák egyszerűen alkalmazhatóak és gyorsak, ezért egy jó választás lehet őket osztályozóként használni az ajánlórendszerekhez [4].

A döntési fa algoritmusok működésének alapja, hogy szükségük van egy olyan adathalmazra, amely be tudja „tanítani” őket. Tehát ennek az adathalmaznak, amelyet gyakran tanítóhalmaznak is neveznek, az osztályai ismertek, melyek egymást kizáró címkék, kategóriák. Az egyedeket sokszor attribútum-értékvektorokként ábrázolják, amelyeknek ismert az a tulajdonságuk, hogy milyen osztályba tartoznak. Ezekből az egyedekből áll a fa tanítóhalmaza. A döntési fa ennek a halmaznak a segítségével osztályozza a még nem ismert, és az ismert elemeket is, mindkettő esetben helyesen meg kell tudnia becsülni azt, hogy az adott egyed melyik osztályba fog tartozni. Ez akkor kerül meghatározásra, ha a felépített fában fentről-lefelé haladva elérünk egy levelet, hiszen az egymást kizáró osztályok a fa leveleihez vannak hozzárendelve [5]. A levelekig a fa gyökerétől a csomópontokon keresztül jutunk el, amelyek egy-egy elágazást valósítanak meg. A csomópontok kialakítása az egyedek különböző attribútumai alapján történnek, a tanító adatok halmazát legjobban felosztó attribútum fogja a fa gyökerét alkotni. Ezután a felosztott adatokat is tovább fogjuk osztani, és ezzel elindul egy rekurzív folyamat, amely akkor fejeződik be, ha az adott ágon az összes maradék adat egy osztályba sorolható [6].

Mint korábban említésre került, az ajánlórendszerek bemeneti adatainak egy része a felhasználóra vonatkozik [3]. Ezeket bekérhetjük valós időben, így aktuálisak lesznek, és akár el is tárolhatjuk őket ahhoz, hogy a későbbiekben felhasználjuk, hogy minél több adat álljon a rendszernek a rendelkezésére. Másik bemeneti adatai az ajánlórendszereknek az egyedekre vonatkozó információk [3]. Ezeknek az adatoknak egy részét érdemes lehet tárolni valahol a programon kívül annak érdekében, hogy a

program futása után is megmaradjanak, illetve lehetnek viszonylag állandó jellegű adatok is, mint pl. amelyek az egyedekre vonatkoznak. Ezeket az információkat érdemes lehet egy adatbázisban tárolni, hiszen nagyméretű adathalmazok is lehetnek a bemeneti adatok [4].

A szakdolgozatomban egy filmekre vonatkozó ajánlórendszert fejlesztettem C# nyelven, amely filmeket ajánl a felhasználó számára az általa adott bemeneti adatok felhasználásával. A felhasználónak a filmek különböző tulajdonságaira vonatkozó prioritásai alapján teszi fel a kérdéseket, és súlyozza a felhasználó által ezekre a kérdésekre adott válaszokat. A teljes adathalmazból véletlenszerűen meghatározásra kerül egy 20 elemű tanítóhalmazt, amelyben a halmaz minden eleme kap egy lebegőpontos pontszámot, amelyek a kérdésekre adott válaszokkal kerülnek kialakításra. A felhasználó számára fontosabbnak, tehát nagyobb prioritásúnak beállított tulajdonságokhoz kapcsolódó kérdésekre adott válaszok nagyobb mértékben fogják befolyásolni a pontszámokat. Végül az így kialakított tanítóhalmaz pontszámait felhasználva megbecsüljük az egész adathalmaz filmjeinek a pontszámait, és a legjobb 5 filmet fogja a program ajánlani a felhasználó számára.

Az információk a programomhoz használt filmekről a „movies.db” adatbázisban kerültek letárolásra. Az SQLite [7] adatbáziskezelő rendszert használtam annak egyszerűsége, hordozhatósága miatt, illetve egy nagy előnye, hogy megbízhatóan tud működni gyakorlatilag szinte minden programozási környezetben.

A felhasznált adatok egy kaggle.com-on megtalálható adatbázisból [8] származnak, amely a TMDb [9]-ről származó adatokkal lett feltöltve, így az általam készített adatbázis adatainak az eredeti forrása a TMDb-n szereplő adatok.

## **2. Ajánlórendszer építéséhez szükséges komponensek bemutatása**

Ebben a fejezetben áttekintésre kerülnek kereső algoritmusok, köztük hagyományosabb és modernebb koncepcióval rendelkezők is. A fejezet további részében olyan komponensek kerülnek bemutatásra, amelyek fontos szerepet játszanak az általam létrehozott ajánlórendszer működéséhez, mint például az adatbázisrendszer, vagy az ML.NET függvénykönyvtár.

### **2.1 Alternatív keresők bemutatása**

Az ajánlórendszereken kívül természetesen vannak egyéb keresők is, mindegyikük más és más elven alapul, és ennek megfelelően más és más célokra is használatosak. A különböző célok mellett ezek a keresők különböző módokon, és helyeken jelenhetnek meg, mint ahogyan az ajánlórendszerek is, hiszen pl. a YouTube-on megtekintett filmek jelentős része ajánlórendszerek ajánlásából származik [2], de emellett felhasználhatóak akár online webáruházak és könyvtárak oldalain is [3].

#### **2.1.1 Kulcsszó alapú keresők**

A kulcsszó alapú keresők a felhasználó által megadott kulcsszavak alapján végzik a keresést, ezekkel a szavakkal keresnek egyezéseket a különböző weboldalakon, dokumentumokban. Ezeknél a keresőknél a precision (pontosság), és a recall (fedés) értékei egyaránt alacsonyak. A precision azt mutatja meg százalékosan, hogy hány olyan találat volt, amely a felhasználó számára az adott keresés szempontjából értékes információkat tartalmazott. A recall azt mutatja meg százalékos formában, hogy mennyit sikerült megtalálni az összes olyan cikkből, dokumentumból, amelyek az adott keresés szempontjából relevánsak. A legnagyobb problémájuk, hogy rengeteg esetben nem képesek releváns találatokat biztosítani, mivel nem ismerik a keresett szavaknak a jelentéseit, ezáltal nem képesek kontextusba helyezni őket. Ez különösen a többjelentésű szavaknál, illetve a szinonimáknál okozhat jelentős problémát. A weboldalak sokszor bizonyos kulcsszavakat tudatosan azért helyeznek el az oldalon, hogy a kulcsszó alapú keresőmotorokkal egyszerűbben, gyorsabban megtalálható legyen az adott oldal [10].

#### **2.1.2 Szemantikus keresők**

A szemantikus keresők ontológia segítségével garantálják, hogy a szavaknak a kapcsolatai és jelentései alapján releváns találatok kerüljenek megjelenítésre. A szemantikus keresők a szemantikus webet használják keresési térként, amely egy olyan jól definiált és jól strukturált információkat tartalmazó változata a webnek, ahol az egyes cikkek, weboldalak különböző szemantikus webes nyelvi elemekkel (RDF, OWL) egészülnek ki, amelyekkel kifejezhetőek a különböző kapcsolatok és összefüggések az egyes források és kulcsszavak között. A szemantikus keresők ezeknek az

információknak a használatával képesek bonyolult keresésekre is releváns találatokat adni [10].

Jelentős eltérés a kulcsszó alapú és a szemantikus keresők között az időtartam. A kulcsszó alapú keresők esetében az eredmények rendezése időigényes tud lenni a felhasználó számára, ezzel szemben a szemantikus keresők rövid idő alatt releváns találatokat tudnak adni, amelyek nem szorulnak rendezésre [10].

### **2.1.3 NLP (Természetes nyelvfeldolgozás)**

A természetes nyelvfeldolgozás (Natural Language Processing vagy röviden NLP), egy olyan tudományterület, amely azt kutatja, hogy hogyan lehet a gépeket az emberi nyelven végzett kommunikáció, leírt szövegek értelmezésére felhasználni. Ezek a keresők az NLP technológia segítségével olyan találatokat képesek biztosítani, amelyek a keresett kulcsszavakat, kifejezéseket nem feltétlenül tartalmazzák, viszont ennek ellenére olyan adatokat tartalmaznak, amelyek kapcsolódhatnak azokhoz. A gépi tanulás ma már teljes mondatok elemzésére is használható, ezáltal a keresett kifejezésben lévő kulcsszavakat a mondat többi részét elemezve képes kontextusba helyezni. Ennek megvalósításához Open IE (Open Information Extraction) technológiát használnak, melynek lényege, hogy az alanyokat, állítmányokat, tárgyakat kinyerik a mondatból, így ezen információk ismeretében könnyebben értelmezhető, hogy mi történik az adott mondatban, mik az aktuális körülmények [11].

Az NLP alapú keresők is küzdenek a többértelműség problémájával, ezeknél is megvan az esély arra, hogy a kereső félreértelmezi a felhasználó által megadott kifejezést, különösen a többjelentésű szavaknál [12].

### **2.1.4 Faceted kereső**

A faceted keresők, vagy más néven gyakorlatilag a szűrők koncepciója az, hogy a felhasználó különböző szempontok, állítások szerint kereshet egy adathalmazban. A felhasználó szemszögéből ez úgy néz ki, hogy különböző kategóriákat lát, amelyeket ki tud választani, és ez által ezek korlátfeltételekként jelennek meg a keresésben, tehát a találatok listája leszűkül azokra az egyedekre, amelyek az adott kategóriába sorolhatóak, rendelkeznek a kiválasztott tulajdonsággal. A szűrő egy-egy kérdése egy állítás (predikátum) szempontjából vizsgálja az egyedeket (pl. ha az adathalmaz egyedei emberek, akkor vizsgálhatóak nem vagy foglalkozás szempontjából). Ezekhez a szempontokhoz többféle válaszlehetőség tartozhat, tehát az egyedek többféle értéket vehetnek fel az adott szempontok szerint vizsgálva, ezért a szűrő működéséhez az adathalmaznak predikátum-értékpárokat kell tartalmaznia [13].

A kategóriák kialakításának a módja történhet hagyományos módon, vagy ha szemantikával ötvözik, akkor ontológia segítségével is. A kategóriák kialakításával kapcsolatban egy olyan probléma merülhet fel, hogy a felhasználók lehetséges, hogy máshogy alakítanak ki őket, hiszen számukra nem túl természetes az aktuális felosztás. Ez lehet amiatt, mert túl általánosan kerültek kialakításra, vagy akár pont az ellenkezője

miatt is, mikor túl szakmaiak lettek a kategóriák, amelyeket pedig a felhasználó már nem ért [14].

### **2.1.5 Kép alapú keresők (CBIR)**

A kép alapú keresők (Content-Based Image Retrieval, röviden CBIR) kevésbé sorolhatók a hagyományos keresők közé. A kép alapú keresők a képeket azoknak a vizuális jellemzői, például szín, alak, textúra alapján keresik vissza. Régebben a képindexelés úgy működött, hogy a képekhez különböző kulcsszavakat, leírásokat rendeltek hozzá a visszakereshetőség érdekében, viszont ez rendkívül lassú, munka és időigényes volt. Erre a problémára jelentettek megoldást a kép alapú keresők, ahol ilyen jellegű munkálatokra már nem volt szükség ahhoz, hogy kereshetőek legyenek a képek [15].

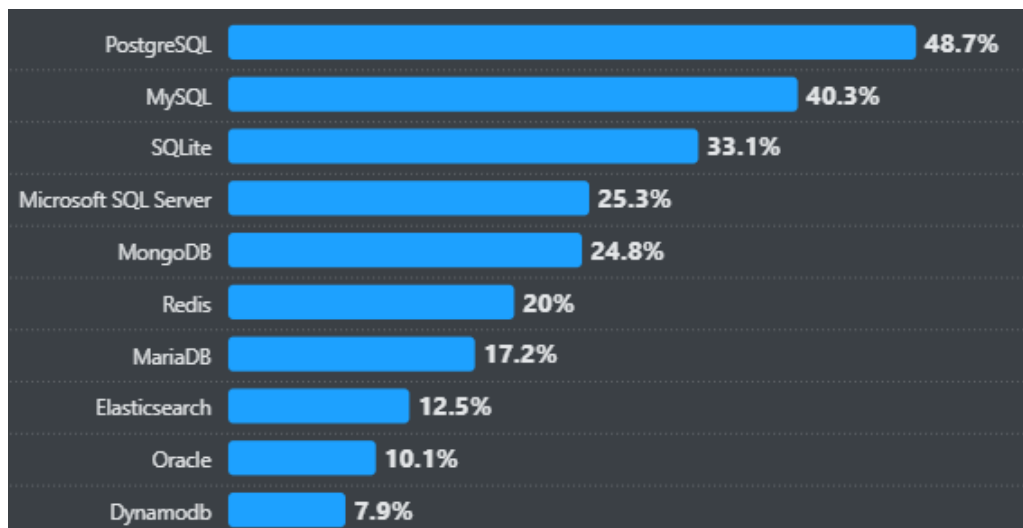
A CBIR folyamata során a képek keresése a következőképpen néz ki: Először megadja a felhasználó azt a képet, amelyhez hasonló képeket szeretne keresni. A kereső elemzi ezt a képet a vizuális tulajdonságai alapján, majd az összes képével ugyanezt teszi annak az adatbázisnak, amelyben keresünk. Ezután a kereséshez használt kép jellemzőit összeveti az adatbázis képeinek jellemzőivel, és a találatok között azok a képek fognak szerepelni, ahol vizuálisan hasonló eredményt kapunk [15].

## **2.2 SQLite bemutatása**

Az SQLite [7] 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, kevés függősége van, és nagy megbízhatóságú [16].

Az SQLite a világ egyik leggyakrabban használt adatbázis-motorja [18]. A cross-platform tulajdonságának köszönhetően számos különböző eszközön, köztük mobiltelefonokban, televíziókban, játékkonzolokban, kamerákban, autókban, és a legtöbb számítógépen is megtalálható valamilyen formában, vagy beépítve, vagy pedig olyan alkalmazások részeként, amelyeket az emberek nap mint nap használnak ezeken az eszközökön [17].

A StackOverflow 2024-es felmérése [18] 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. ábra).



1. ábra: A StackOverflow 2024-es felmérése különböző adatbázis motorok elterjedtségéről [18]

A listában csak a PostgreSQL és a MySQL előzik meg, viszont ezek nehezen tekinthetők az SQLite konkurensainak, 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 [17].

Mivel a szakdolgozatom középpontjában egy helyi, C# program áll, így a kliens/szerver adatbázis motorok helyett egy kisebb méretű helyi adatbázisra 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 [17].

Mivel egy C# nyelvű programról van szó, ezért az adatbázismotor kiválasztásánál egy fontos kitétel volt, hogy C# nyelvvel könnyen, és jól használható legyen. A *System.Data.SQLite* egy ADO.NET szolgáltatás az SQLite-hoz [19]. Az SQLite adatbázismotor 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.

## 2.3 Az adatbázis felépítése

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

### 2.3.1 Az adatbázis szerkezete

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

- ID:
  - Típus: INTEGER
  - Leírás: A film egyedi azonosítója, ez az 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) felsorolásszerűen, 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 játékidéje percben megadva.
- Gender\_of\_the\_protagonist:
  - Típus: INTEGER
  - Leírás: A főszereplő nemét 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 bevételeinek összege amerikai dollárban megadva.

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

- **ID:**
  - Típus: INTEGER
  - Leírás: Az adott ország egyedi azonosítója, ez az elsődleges kulcs.
- **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 az elsődleges kulcs.
- **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 az elsődleges kulcs.
- **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 az elsődleges kulcs.
- 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 az elsődleges kulcs.
- 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, idegenkulcs, amely a *Movies* tábla ID mezőjére mutat.
- Countries\_ID:
  - Típus: INTEGER
  - Leírás: Az adott ország egyedi azonosítója, idegenkulcs, 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, idegenkulcs, amely a *Movies* tábla ID mezőjére mutat.
- Directors\_ID:
  - Típus: INTEGER
  - Leírás: Az adott rendező egyedi azonosítója, idegenkulcs, 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, idegenkulcs, amely a *Movies* tábla ID mezőjére mutat.
- Genres\_ID:
  - Típus: INTEGER

- Leírás: Az adott műfaj egyedi azonosítója, idegenkulcs, amely a *Genre* tábla ID mezőjére mutat.

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

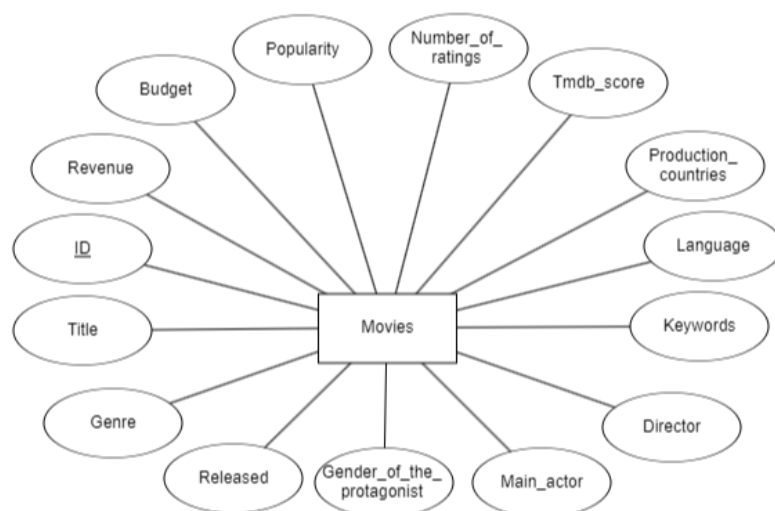
- **Movie\_ID:**
  - Típus: INTEGER
  - Leírás: Az adott film egyedi azonosítója, idegenkulcs, amely a *Movies* tábla ID mezőjére mutat.
- **Keywords\_ID:**
  - Típus: INTEGER
  - Leírás: Az adott kulcsszó egyedi azonosítója, idegenkulcs, amely a *Keyword* tábla ID mezőjére mutat.

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

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

### 2.3.2 Az adatbázis ER modellje

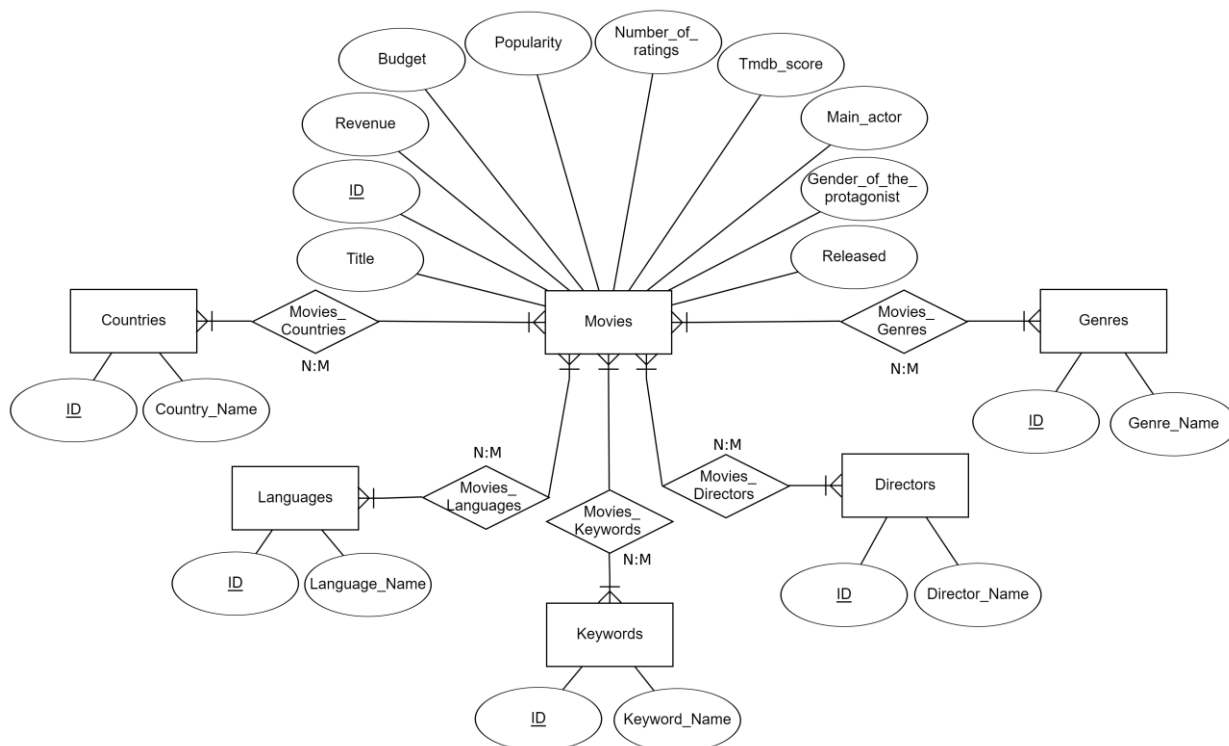
Az adatbázisok szerkezeti felépítésének szemléltetésére két gyakran használt módszer van, ezek közül az egyik az ER modell. A diagramon a táblákat téglalapok jelölik, benne a nevükkel, a táblák mezőit pedig ellipszisek, amelyek belsejében szintén megtalálható a nevük is. A mezők vonalakkal vannak összekötve azzal a táblával, amely tartalmazza őket. Az adatbázis kiindulási ER modelljén jól látszik, hogy az adatbázis csupán egyetlen táblát tartalmaz, és az összes mező ehhez a táblához kapcsolódik. Az ID mező neve aláhúzással van megjelölve, ez azt szemlélteti, hogy a táblának ez az elsődleges kulcsa (2. ábra).



2. ábra: A movies.db adatbázis kiindulási, lebontás előtti ER modellje

A *Movies* tábla adatainak tartalmát vizsgálva a legtöbb egy egyszerű értéket tartalmaz, azonban vannak közöttük olyan mezők is, amelyek több értéket is tartalmaznak, egymástól vesszővel elválasztva. Ezeket az értékeket ebben a formában nem lehet hatékonyan felhasználni, mivel a mezők különböző értékeit nem tudjuk külön-külön vizsgálni, csak egyben. Például ha egy filmben 3 nyelven is megszólalnak, ezáltal a nyelve „Angol, Magyar, Német”, akkor ebben a formában erről a filmről csak ennyit tudunk elmondani. Viszont ha külön-külön arról szeretnénk információt kapni, hogy az adott filmben beszélnek-e „Angol” vagy „Magyar” nyelven, akkor az adatbázisban nincs olyan információ, ami konkrétan erre a kérdésre adna választ, ez pedig megnehezíti az adatok, jelen esetben filmek ezen tulajdonságokon alapuló egymással történő összevetését.

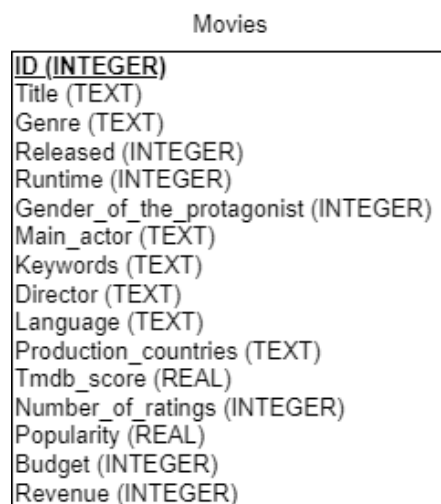
Az érintett mezők tartalmait lebontva az adatbázis szerkezete nagyban megváltozik, melyet a lebontás utáni ER modell jól szemléltet (3. ábra). Az érintett mezők már nem csak közvetlenül a *Movies* táblához tartoznak (az adatbázis szerkezetében továbbra is jelen vannak ott is, de az ábra bonyolultságának elkerülése érdekében nincsenek ábrázolva), hanem külön táblákká alakultak át, és ezeket a közöttük lévő N:M típusú kapcsolatok kapcsolják a *Movies* táblához. Mivel a kapcsolatok mindkét irányba több-több (N:M), így ennek tényleges megvalósításához kapcsolótáblák létrehozása szükséges, viszont ez nem látszik az ER diagramon.



3. ábra: A movies.db adatbázis lebontás utáni ER modellje

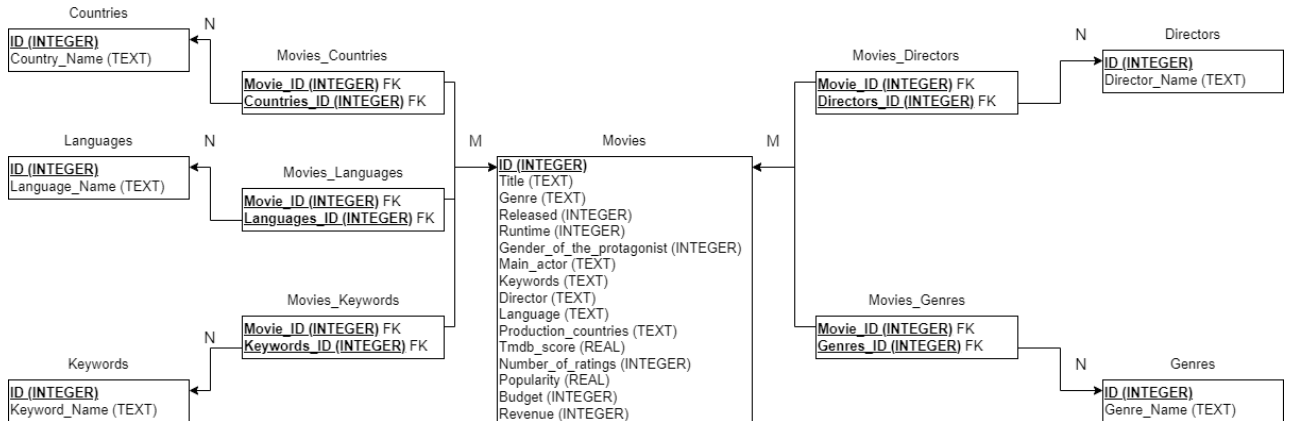
### 2.3.3 Az adatbázis relációs modellje

Az adatbázisok szerkezeti felépítésének másik gyakori módszere a relációs modell. Az adatbázis szerkezetének az állapotát az előbbieken részletezett lebontás folyamatát megelőzően a kiindulási relációs modell mutatja meg (4. ábra). A relációs modellben is téglalapok jelzik a táblákat, viszont itt a mezők a téglalapba beleírva helyezkednek el. A kiindulási relációs adatmodell annyival tartalmaz több információt az adatbázisról a kiindulási ER modellel (2. ábra) szemben, hogy itt az is jelölve van, hogy a különböző mezők milyen típusú adatokat tartalmaznak.



4. ábra: A movies.db adatbázis kiindulási, lebontás előtti relációs modellje

Az adatbázis szerkezetének a lebontás folyamata utáni állapotát szemléltető relációs modellje már részletesebb képet ad az adatbázisról (5. ábra). Itt már megjelennek a kapcsolótáblák, és azok mezői is, továbbá szintén megjelennek a mezőkben tárolt adatok típusai is.



5. ábra: A movies.db adatbázis lebontás utáni relációs modellje

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 ilyen jellegű több-több kapcsolat miatt van tehát szükség a kapcsolótáblákra minden lebontott mező esetében.

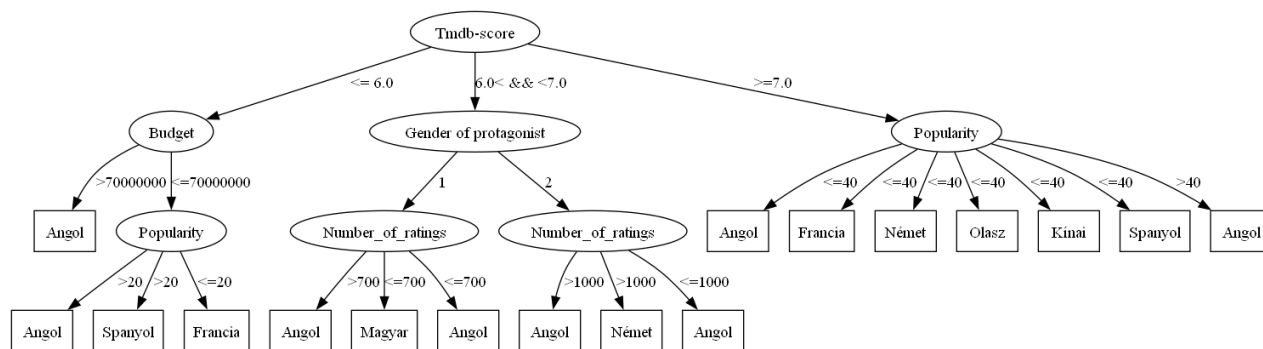
A táblák közötti kapcsolatok mindkét irányba kötelezőek: Minden filmnek van műfaja, gyártási országa, rendezője, stb., ezért ez kötelező jellegű kapcsolat. A lebontott tulajdonságok esetében, tehát pl. a műfajokat, rendezőket, kulcsszavakat, stb. tartalmazó táblákban alapvetően előfordulhatnak olyan adatok, amelyekhez nem rendelünk filmet, hiszen egyik film sem rendelkezik ilyen tulajdonságokkal. Viszont a program felépítéséből adódóan ez nem lehetséges, hiszen ezeket a táblákat az adatbázisban lévő filmek adatait használva töltjük fel, és amennyiben ezek közül bármelyik film, vagy annak adata törlődik, akkor az összes ilyen jellegű tábla tartalmát frissítjük. Tehát pl. a *Genres* táblát nézve annak minden egyes műfajához fog legalább egy film kapcsolódni a *Movies* táblában.

## 2.4 Kézzel készített döntési fa a film nyelvére vonatkozóan

A szakdolgozatom témája a kérdőív alapú adaptív ajánlórendszerek, melyekhez szorosan kapcsolódik a döntési fák témaköre is. Az ajánlórendszerek olyan gépi tanulási technikák, amelyek különböző termékek, szolgáltatások sokaságát tudják javasolni a felhasználó számára a felhasználtól származó információk alapján, ezáltal könnyebbé teszi számára az ő ízlésének megfelelő termékek, szolgáltatások megtalálását [3].

A beérkező információk feldolgozásának egyik módszere a döntési fák, melyek nagy pontossággal képesek osztályozni a különböző információkat, így tökéletes alapot tudnak szolgáltatni az ajánlórendszer döntéseinek meghozatalában [4].

A döntési fák szemléltetésére létrehoztam egy olyan fát, amely kézzel került összerakásra, a leírásához JSON formátumot választottam (6. ábra). Azért a JSON formátumra esett a választásom, mert egy olyan platformfüggetlen formátumról van szó, amelyet szinte minden programozási nyelv támogat. Egyszerű a szintaktikája, a használatával leírt adatok, modellek egyszerűen feldolgozhatóak más programozási nyelvek segítségével, továbbá érthetőek az emberek és gépek számára is egyaránt [20].



6. ábra: A film nyelvére vonatkozó kézzel készített fa ábrázolása

A fa egy 20 elemű tanító halmaz segítségével került kialakításra, 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. A döntési fa JSON nyelvű leírása az alábbi:

```
{
  "decision_tree": {
    "root": {
      "node": "Tmdb-score",
      "condition": [
        {
          "condition": "<= 6.0",
          "outcome": {
            "node": "Budget",
            "condition": [
              {
                "condition": ">70000000",
                "outcome": "Angol"
              },

```

```

{
  "condition": "<=700000000",
  "outcome": {
    "node": "Popularity",
    "condition": [
      {
        "condition": ">20",
        "outcome": ["Angol", "Spanyol"]
      },
      {
        "condition": "<=20",
        "outcome": "Francia"
      }
    ]
  }
}

```

## 2.5 Kézzel készített döntési fa hatékonysága

A bemutatott döntési fa hatékonyságának mérését az 1. táblázat mutatja be.

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

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

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 különböző 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 1. 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 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 2. táblázat tartalmazza.

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

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

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 megnőveltem. Miután végigment a program a filmekben, 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.

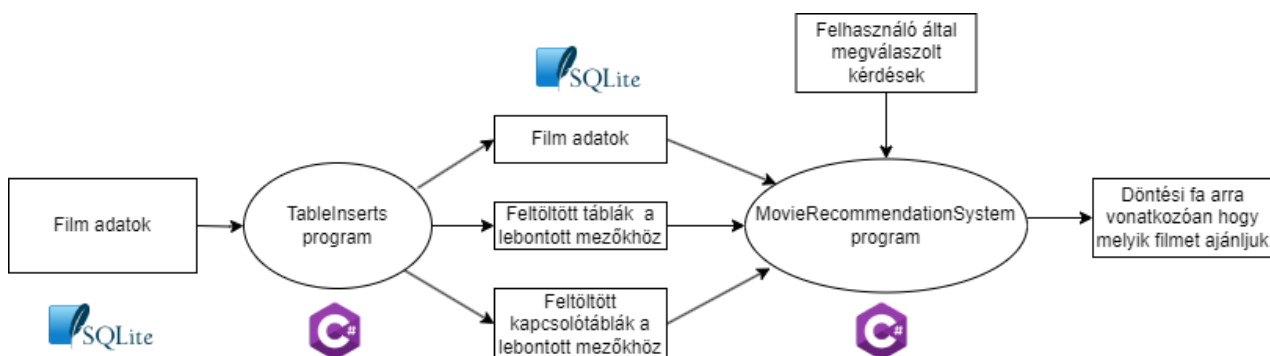
## 2.6 Folyamatábra az adatbázis adatainak feldolgozásáról

Egy folyamatábrával került szemléltetésre az, hogy hogyan, és milyen eszközökkel jutunk el az adatbázisba összegyűjtött filmektől a felhasználó számára ajánlott filmekig (7. ábra). A folyamatban a feldolgozóegységek a C# nyelven írt programok. Azon belül a *TableInserts* egy egyszerű konzolos, tehát grafikus felülettel nem rendelkező program,



amely a korábbiakban részletezett mezők lebontásáért felelős, tehát gyakorlatilag egy segédprogramként szolgál a fő program számára, hiszen a feldolgozáshoz szükséges formába hozza a filmek adatait. A TableInserts program részletes, kód szintű bemutatása a melléklet „A” fejezetében található meg. A lebontással új táblák jelennek meg az eredeti *Movies* tábla mellett, ezekkel egészül ki az eredeti, SQLite formátumban tárolt adatbázis. A fő grafikus program, amely az adatok feldolgozását, kezelését, megjelenítését, illetve magát az ajánlórendszert is megvalósítja, már a TableInserts által átalakított, előkészített adatbázist kapja meg bemenetként. A fő program, tehát a „MovieRecommendationSystem” felépítésének részletes bemutatását a melléklet „B” fejezete tartalmazza.

Másik bemeneti forrása a felhasználó által adott információk, a megválaszolt kérdések, hiszen ezek felhasználásával, feldolgozásával fogja elvégezni a program az ajánlott filmekre vonatkozó becslését, amely majd a program kimenetétül fog szolgálni.



7. ábra: Folyamatábra a feldolgozandó adatokról, és a feldolgozóegységekről

## 2.7 ML.NET bemutatása

Ebben a fejezetben egy C#-hoz használható gépi tanulási metódusokat tartalmazó függvénykönyvtárról lesz, szó, az ML.NET-ről [24].

### 2.7.1 A mesterséges intelligencia és a gépi tanulás kapcsolata

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 [21].

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 adat elemzésére, a felismerésekből való tanulásra, majd ezek alapján a döntések meghozatalára [21].

A gépi tanulási algoritmusok teljesítménye idővel javulni tud azáltal, hogy egyre több adattal találkozhatnak amelyeket elemeznek, majd mintákat ismernek fel belőle, amiknek a megtanulásával egyre pontosabb eredményeket tudnak elérni [21].

Maga a gépi tanulási modell alatt azt értjük, amit a program az algoritmus a tanító adatokon történő futtatásából tanul. Minél több adatot használunk, várhatóan annál jobb, pontosabb lesz a modell [21].

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 mpack [22], a Pythonhoz a scikit-learn [23] függvénykönyvtárak használhatóak a gépi tanulás algoritmusainak eléréséhez, az általam is használt C# nyelv esetén pedig az ML.NET [24] függvénykönyvtár tartalmazza ezeket az algoritmusokat.

### **2.7.2 Mi az az ML.NET?**

Az ML.NET [24] 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. [25] Az ML.NET segítségével betanítható egy egyéni modell, vagy akár importálhatóak előre betanított TensorFlow- és ONNX-modellek [26].

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 [26].

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 [25]. 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.

## **2.8 Hogyan néz ki egy ML.NET alkalmazás?**

Egy ML.NET program általános felépítését a következő alpontok részletezik.

### **2.8.1 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 [27].

```
var mlContext = new MLContext();
```

(Az ML.NET dokumentációjából [27] származó kódrészletek)

## 2.8.2 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 [27].

Az adatok IDataView objektumokba kerülnek betöltésre, amelyek tartalmazhatnak különböző típusú számokat, szövegeket, logikai értékeket, vagy akár vektorokat és számokat tartalmazó tömböket 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 [27].

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

Amennyiben már memóriában lévő, változókból, tömbökből származó adatokat szeretnénk betölteni, akkor arra is lehetőségünk van [27].

```
IDataView trainingData =  
mlContext.Data.LoadFromEnumerable<SentimentInput>(inMemoryCollection);
```

## 2.8.3 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 [27]. A szöveggént tárolt értékek egy részét 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 [28]. 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 egész szám, 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 alából 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.

## 2.8.4 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 [27]. 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 [29]:

- **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 [29].
- **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 [29].
- **Mátrix-faktorizációk:** A collaborative filtering rendszerekhez használatos algoritmusok. [29] A collaborative filtering, vagy magyarul együttműködő szűrő rendszerek egy személynek különböző 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 [30].
- **Meta-algoritmusok:** Ezen algoritmusok segítségével bináris osztályozókból többosztályos osztályozók készíthetők [29].
- **K-közép:** A klaszterezés (beleértve a K-Közép) egy felügyelet nélküli tanítási mód [31] [29].
- **Fő komponens analízis:** Hibakeresésre, anomáliák felderítésére használatos algoritmusok [29].
- **Naiv Bayes osztályozó:** Többosztályos osztályozók esetében érdemes használni ezen algoritmusokat abban az esetben, mikor a jellemzők függetlenek egymástól, illetve kis mennyiségű adat áll rendelkezésünkre a modell betanításához [29].
- **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 [29].
- **Tartó vektor gépek (Support Vector Machine, röviden SVM):** Felügyelt gépi tanulási algoritmusok, a jellegvektorok terét igyekeznek szeparátor görbék, felületek segítségével szétválasztani (magasabb dimenziós terekben is). Velük

kapcsolatos megoldandó probléma, hogy hogyan lehet őket hatékonyan használni nagyobb adathalmazok esetében [29].

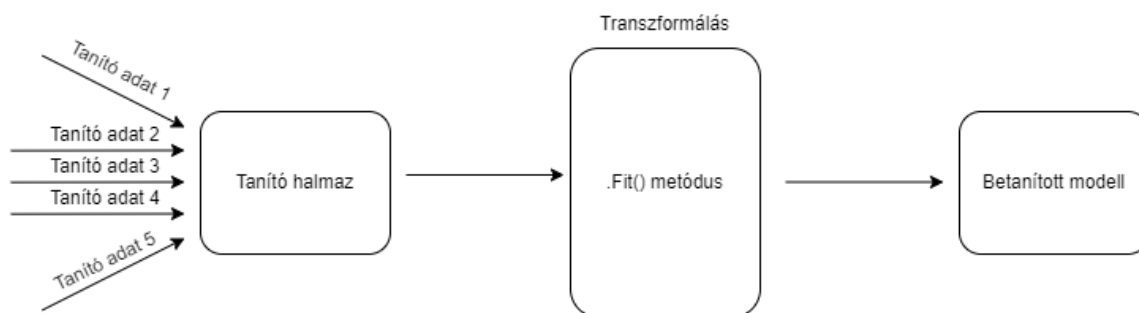
- Legkisebb négyzetek módszere: A lineáris regressziónak ez egyik leggyakrabban használt felhasználási módszere. Veszteségfüggvénnyel méri a modell hibáját, azaz azt, hogy mennyire tér el a modell által előrejelzett érték a ténylegesen megfigyelt értéktől [29].

### 2.8.5 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 paraméterbecslés, a modell illesztése és hangolása történik [29].

A modell betanításának folyamatát a 8. ábra szemlélteti. A folyamatban a tanító adatok lesznek a bemeneti adatok, majd a modell tanítása, mint egy transzformátor jelenik meg. A transzformálás után kimenetként a betanított modellt kapjuk, amellyel előrejelzéseket, becsléseket készíthetünk a még ismeretlen adatokra vonatkozóan [29].

```
ITransformer model = pipeline.Fit(trainingData);
```



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

### 2.8.6 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 osztályozást 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 [32]:

- 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 [32]. 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 [33]. A kapott érték minél közelebb van 1-hez, annál jobb, hiszen annál pontosabb a vizsgált modell [32].

- **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 [32].
- **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 log-veszteség értéke, a modellünk annál pontosabb becslésre képes [32].
- **Log-veszteség csökkentése:** A log-veszteség csökkentésénél a véletlenszerű találgatáshoz 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 [32].

A Microsoft ML.NET keretrendszerhez kapcsolódó, a modell kiértékeléséről szóló dokumentációja [32] szerint ha a felsorolt hatékonyságmérési módszerek közül csak egyet lehetne választani, akkor a legtöbb esetben mindenképpen a mikro-pontosságot érdemes.

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 [32]:

- **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.

- Abszolút veszteség: Szintén arra vonatkozó mérőszám, hogy a becült értékek mennyire állnak közel a valóshoz. Minden egyes értékre kiszámításra kerül a hiba mértéke, amely a becült érték és a tényleges közötti abszolút távolság. Végül ezekből az értékekből számolunk egy átlagot, ez lesz a kapott abszolút veszteség. Minél közelebb van a szám a 0-hoz, annál pontosabb a modell.
- 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.
- RMS-veszteség: A négyzetes eltérés négyzetgyöke, ezáltal könnyebben értelmezhetővé teszi a hibamérést. Minél közelebb van a 0-hoz, annál pontosabb a vizsgált modell.

### 2.8.7 A modell kimentése, használata

Az elkészített modellek kimentésére is lehetőséget kínál az ML.NET keretrendszer. Alap esetben 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 [34].

A modell mentéséhez a Microsoft ML.NET dokumentációja [34] 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 dokumentációjából [34] származó kódrészlet)



A kimentett modell visszatöltéséhez egy `DataViewSchema` és egy `ITransformer` típusú változóra van szükségünk: [34]

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

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

## 2.9 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 lefusszon 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 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 3. táblázatban láthatóak.

3. táblázat: A különböző algoritmusok mérései során kapott  $R^2$  értékek

	1.	2.	3.	4.	5.
Fast tree	-69,63	-69,63	-69,63	-69,63	-69,63
Fast tree Tuned	0,172	0,128	0,162	0,117	0,184
Sdca	0,07	0,08	0,11	0,098	0,103
Sdca Tuned	0,143	0,145	0,148	0,14	0,152
Lbfgs Poisson R.	-0,17	-0,17	-0,172	-0,17	-0,17
Lbfgs Poisson R. Tuned	-0,190	-1,822	-0,163	-1,586	-0,846
Gam	-69,63	-69,63	-69,63	-69,63	-69,63
Gam tuned	-69,63	-69,63	-69,63	-69,63	-69,63
Online Gradient Descent	-15,32	-11,29	-12,342	-8,227	-14,286
Online Gradient Descent tuned	-5,74	-5,716	-5,08	-4,399	-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.

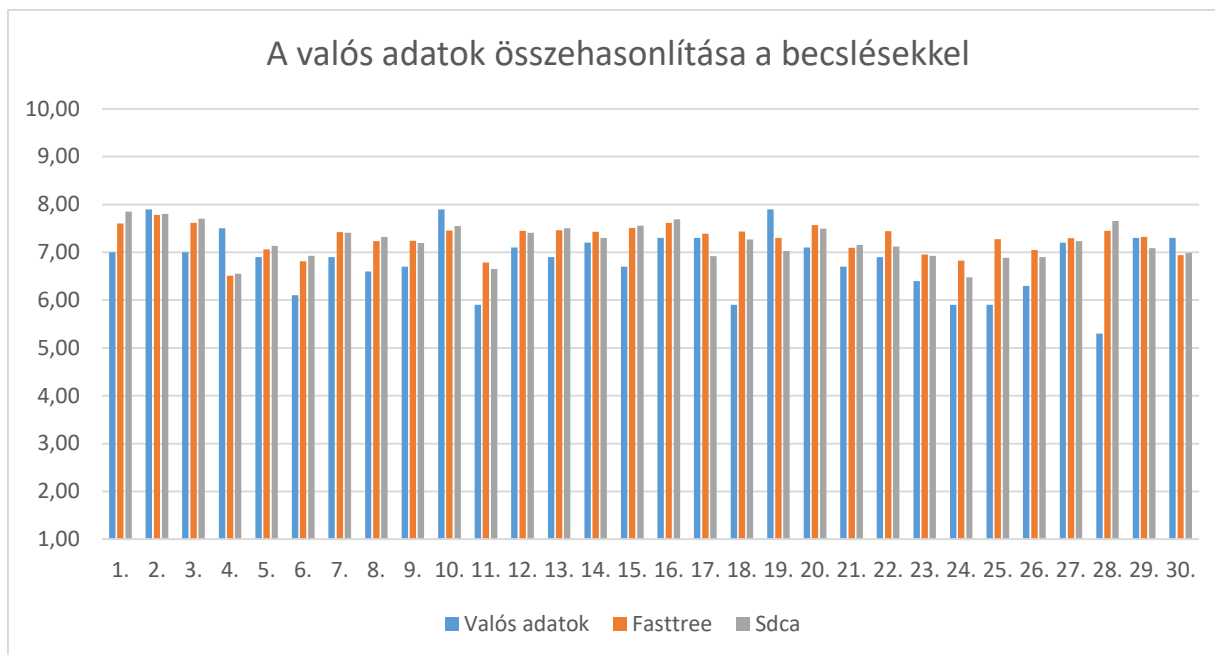
Az  $R^2$  mérőszám értéke mínusz végtelentől 1-ig terjedhet. Minél magasabb az érték, tehát minél közelebb van 1-hez, annál hatékonyabban működik a modell. Ha negatív értéket vesz fel, az azt jelöli, hogy a modell rosszabbul működik mint a véletlen találgatás. Ezen információk ismeretében az alábbi 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 0,1545-et, a Fasttree esetén 0,1691-et kapunk, tehát az  $R^2$  mérőszámokat figyelembe véve a Fasttree algoritmus működött az összes közül a leghatékonyabban.

A két legjobban működő algoritmus becsléseit oszlopdiagram használatával ábrázoltam a valós adatok mellett (9. ábra).



9. ábra: A Fasttree és Sdca becslései oszlopdiagrammal ábrázolva a valós adatok mellett

Egy 30 filmből álló tartomány értékeit ábrázolja a diagram. Mint az ábra is mutatja, kék színnel vannak ábrázolva a valós adatok, szürke színnel az Sdca algoritmus által becsült adatok, és narancssárga színnel pedig a Fasttree algoritmus által becsült értékek minden egyes film esetén. Az ábrából látszik, hogy az esetek nagy részében mindkét becslés valamivel magasabb értéket becsült, mint a valós adatok. A kiugróan alacsony értékeket nem tudták minden esetben helyesen megbecsülni, ami érthető abból a szempontból, hogy az előrejelzéshez a filmeknek csak a kulcsszavai és műfajai kerültek felhasználásra, azokból pedig nem feltétlenül lehet az összes esetben pontosan következtetni arra, hogy milyen lesz a film fogadtatása, amelyet gyakorlatilag a TMDB pontszám tükröz.

A fenti eredmények alapján kijelenthető, hogy az  $R^2$  mérőszám segítségével mérve az említett problémára a Fasttree algoritmus adta a legpontosabb becsléseket az adott paramétertartományokban megtalált legjobb paraméterértékekkel. Ennek ellenére a lineáris regresszió segítségével végzett vizsgálatnál kiderült, hogy az Sdca által adott becslések lineáris kapcsolatban jobban illeszkednek a valós adatokhoz. Az Sdca algoritmus alapvetően lineáris regressziót használ az eredmények megbecsléséhez [35], így érthető, hogy az eredményeit ábrázoló egyenes meredeksége miatt követi jobban a valós adatokét.

A 2 algoritmus pontossági 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 másikonál.

## 2.10 Gyakorlás céljából készített becslő algoritmusok hatékonyságának összehasonlítása

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 le mérésre a következő 2 becslésnek a hatékonysága is, amely a film rendezőjére, és a főszereplő nemére vonatkozott.

A film rendezőjére vonatkozó döntési fánál a 4. táblázatban szereplő hatékonysági számokat kaptam.

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

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

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 az 5. táblázatban található eredményeket kaptam.

5. táblázat: A film rendezőjére vonatkozó becslő algoritmus mérési eredményei

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

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.

### 3. Az implementált program ajánlórendszerének felépítése

Az elkészített programom az ajánlórendszer funkciója mellett több más funkcióval is rendelkezik, mint például az adatbázis filmjeinek beolvasása, megjelenítése, a teljes adatbázis cseréje, új filmek hozzáadása, meglévők törlése, vagy módosítása. Továbbá ahhoz, hogy az adatok olyan formába kerüljenek, amelyekkel hatékonyan lehet dolgozni, a `MovieRecommendationSystem` főprogramba integrált `TableInserts` segédprogramra is szükség van. A programoknak, és az előbbiekben felsorolt funkciók megvalósításának a kódszintű bemutatása a mellékletként csatolt „Programok kódszintű bemutatása” dokumentum „A” és „B” fejezetében találhatóak. Ebben a fejezetben a programnak csak a szakdolgozatom témájának szempontjából a legfontosabb rész, az ajánlórendszer részének felépítése kerül bemutatásra.

#### 3.1 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 esetben 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 következőképpen 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 felsorolást 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 típusra alakítva,
- 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 képernyőtervét a 10. ábra szemlélteti.

10. ábra: A Prioritásválasztó ablak képernyőterve

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 listbox-ot 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 (minél kisebb a prioritási szám, annál nagyobb prioritással vesszük az adott jellemzőre adott választ), a legalján lévő elem pedig a legkisebbel.

A „>” gomb algoritmus a következő:

```

private void MoveSelectedItem()
{
    if (listBoxOriginal.SelectedItem != null)
    {
        PriorityListItem selectedItem =
        (PriorityListItem)listBoxOriginal.SelectedItem;
        originalList.Remove(selectedItem);
        selectedList.Add(selectedItem);
        RefreshListBoxes();
    }
}

```

A többi elemmozgatásra használt gomb algoritmusai 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;
    }
}

```

A „Move Up” és a „Move Down” gombok a következő logika szerint működnek, jelen esetben a felfelé mozgatást bemutatva: Először eltároljuk a kiválasztott elem indexét, majd ellenőrizzük, hogy ez az index nagyobb-e, mint 0. Ha az elem nem a lista tetején található (index nem 0), akkor átmenetileg eltároljuk az elemet, amit mozgatni szeretnénk, majd töröljük a listából az eredeti helyéről. Ezt követően eggyel kisebb indexű pozícióba illesztjük vissza az elemet, így a listában eggyel feljebb kerül. Miután elvégeztük a módosítást, frissítjük a listboxokat, és visszaállítjuk a kijelölést az új helyen lévő elemre, hogy az továbbra is ki legyen jelölve.

```

/// <summary>
/// A kiválasztott elem felfelé mozgatása a listboxban a magasabb prioritás
/// érdekében
/// </summary>
private void MoveUp()
{
    int selectedIndex = listBoxSelected.SelectedIndex;
    if (selectedIndex > 0)
    {
        var item = selectedList[selectedIndex];
        selectedList.RemoveAt(selectedIndex);
        selectedList.Insert(selectedIndex - 1, item);
        RefreshListBoxes();
        listBoxSelected.SelectedIndex = selectedIndex - 1;
    }
}

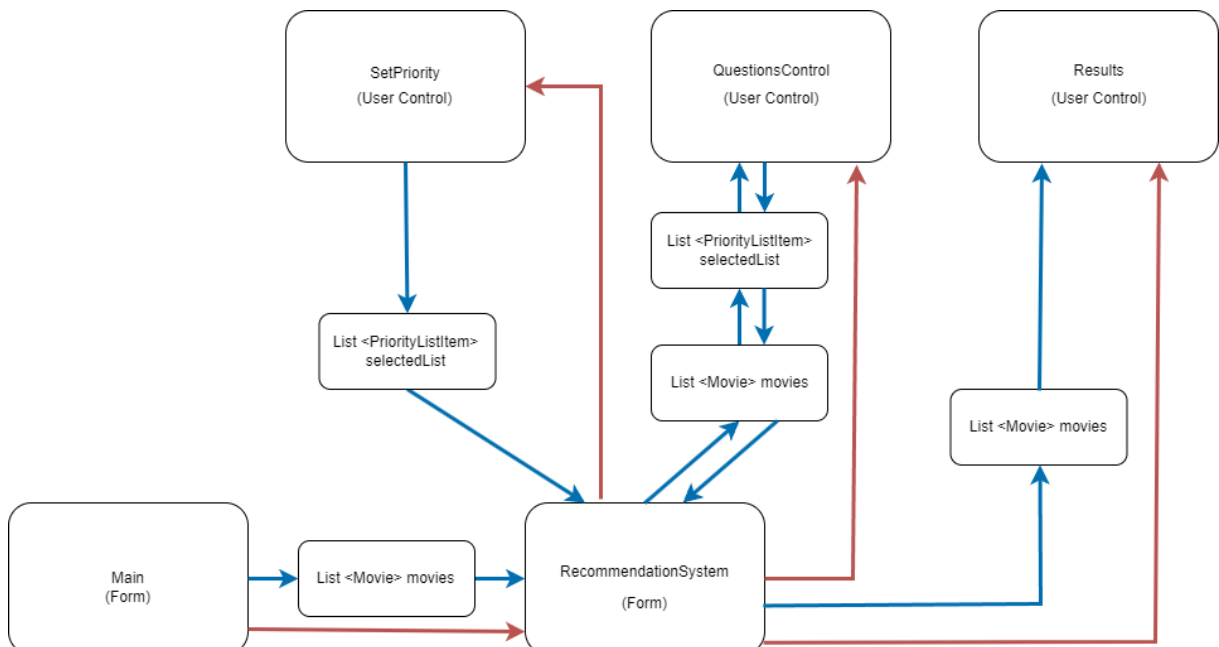
```



## 3.2 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.



11. ábra: A *RecommendationSystem* Form-ba betöltött User Controlok kapcsolata



A 11. 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.

### 3.2.1 Kérdések kialakítása

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 típusra alakítva 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* osztályszintű változóval oldjuk meg, amelyet csak a „Next” gomb megnyomásakor, tehát a következő kérdésre történő továbblépéskor 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;
        }
    }
}

```

### 3.2.2 Tanítóhalmaz kialakítása, feltöltése

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 pontszámánál, 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 tartalmazza, 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 *LearningMovie* 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; }
}

```

### 3.2.3 A kérdések összekapcsolása a jellemzőkkel, és a válaszokkal

A programnak ezen a pontján a kérdések ugyan már megjeleníthetők a prioritási lista tartalmának felhasználásával, és a válaszlehetőségek kiválasztásához szolgáló

checkboxlistbox 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 állapotát a 12. ábra szemlélteti.

12. ábra: A prioritásválasztó képernyő állapota egy adott példára

### 3.2.4 A checkboxokat használó, és több válaszopciót is elfogadó kérdések kezelése

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 filmajánlót,

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.

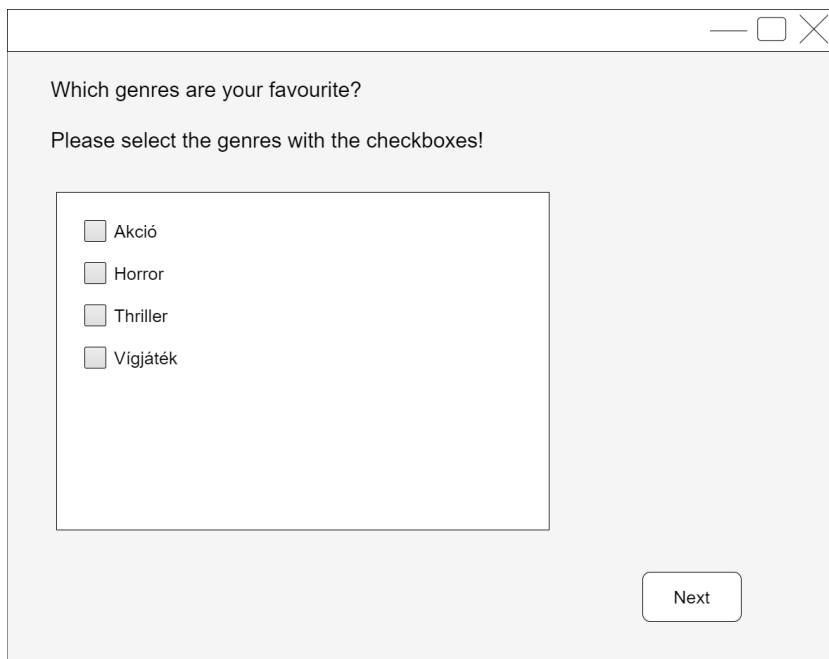
```
private void QuestionsControl_Load(object sender, EventArgs e)
{
    SetScore();
    InitFeaturesForCheckbox();
    CreateLearningData();
    CreateQuestions();
    LoadQuestion();
    LoadAnswers(receivedPriorityList[currentIndex].Id);

    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 13. ábrán látható módon fog kinézni a képernyő.



13. ábra: A példánkban az első kérdés betöltése után megjelenő képernyő terve

A felhasználó a kedvenc műfajainak kiválasztása után megnyomja a „Next” gombot a következő kérdésre lépéshez. Ekkor a „Next” gomb megnyomásához rendelt 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 kérdéstről 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 leellenőrizni, 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 eltüntetjük őket és a hozzájuk tartozó elemeket, ha a következő kérdésre ugrunk.

A *Genre* jellemzőhöz hasonló, checkboxos, több választ is kiválasztható 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 eltüntetni, amennyiben láthatóak.

A megkülönböztetés miatt az igaz/hamis kérdések kezeléséhez is bevezetésre került egy logikai 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 átváltanunk 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 osztályszintű *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 átadjuk még a kiszámított pontszámot is.

A *ModifyScore* különböző verziói a következők:

- *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*,
- *ModifyScoreForTrackbar*: az olyan jellemzőkhöz, amelyekhez kapcsolódó kérdésekre csúszkával kell válaszolni.

A metódusok között csak a bemeneti adatokban, és ez által 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.

### 3.2.5 A csúszkákat használó kérdések kezelése

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 14. ábrán látható képernyő kerül megjelenítésre.

What is the range of years you want to watch movies?

Please set the minimum and maximum values with the sliders, or write them into the boxes!

1950 2024

1950 2024

1996 2016

Next

14. ábra: A példánkban az második kérdés betöltése után megjelenő képernyő terve

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 minimum 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* osztályszintű logikai 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.

### 3.2.6 Az igaz/hamis kérdések kezelése

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 képernyőnek a kérdés betöltése utáni állapotát a 15. ábra mutatja be.

15. ábra: A példánkban az harmadik kérdés betöltése után megjelenő képernyő terve

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* értékekből.

```
/// <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 ebben az esetben a *ModifyScoreForTrueOrFalse* metódust hívja meg, amelynek logikája megegyezik a többi *ModifyScore* típusú metódusával.

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 *QuestionsControl*-ra), ahol a becslött eredményeket megjelenítjük.

### 3.2.7 Az eredmények becslése

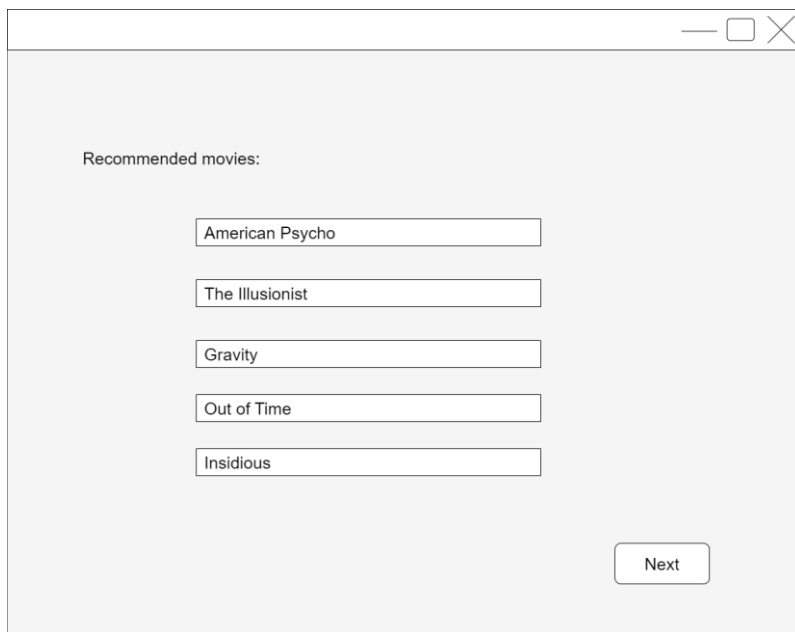
A fő becslés tehát a *MainDecTree* osztály *BuildTree* metódusában található, logikája pedig teljes mértékben megegyezik a *DecTreeForTmdb* osztályban megvalósított, a filmek TMDB pontszámára vonatkozó becsléssel, amely a melléklet B.8-as fejezetében került bemutatásra. Az elkészített fő becslés a szakdolgozatom 2.8-as fejezete alapján készült, amelyben részletezésre kerülnek azok a lépések, illetve összetevők, amelyek egy ML.NET becslési program elkészítéséhez szükségesek.

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. Ezt a képernyőt a 16. ábra mutatja be.

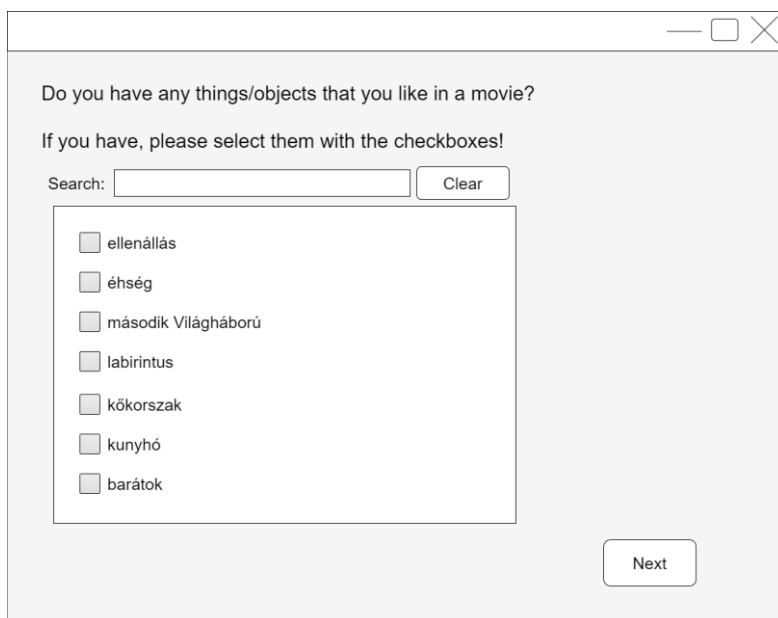




16. ábra: Az ajánlott filmek képernyőterve

### 3.2.8 Kereső megvalósítása a kulcsszavakhoz

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 kereső elhelyezését a 17. ábra szemlélteti.



17. ábra: A kulcsszavak esetében használt kereső

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 aktualizáljuk. 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++)
```

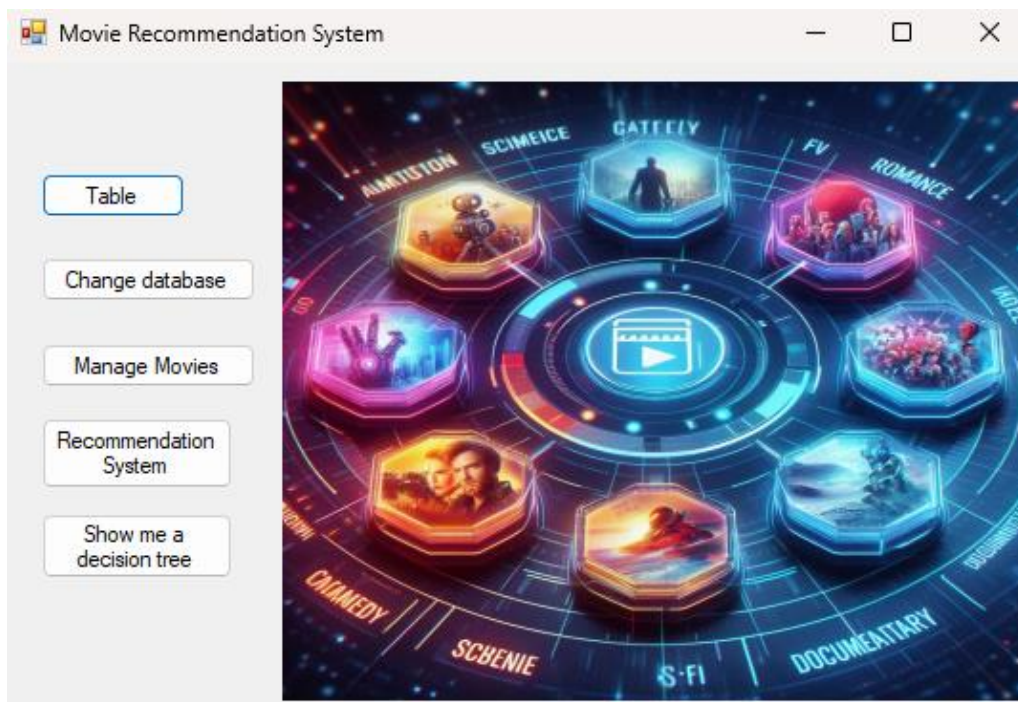
```
{
    itemText = checkedListBox.Items[i].ToString();
    if (selectedFromCheckbox.Contains(itemText))
    {
        checkedListBox.SetItemChecked(i, true);
    }
}
```

## 4. A program használatának a bemutatása

Ebben a fejezetben a kész program felhasználói felületének, működésének a bemutatására kerül sor a felhasználó szemszögéből. Továbbá részletezésre kerül, hogy egy C# nyelven írt program esetében a forráskódból milyen módon lehet a futtatható kódot előállítani.

### 4.1 A program funkciói

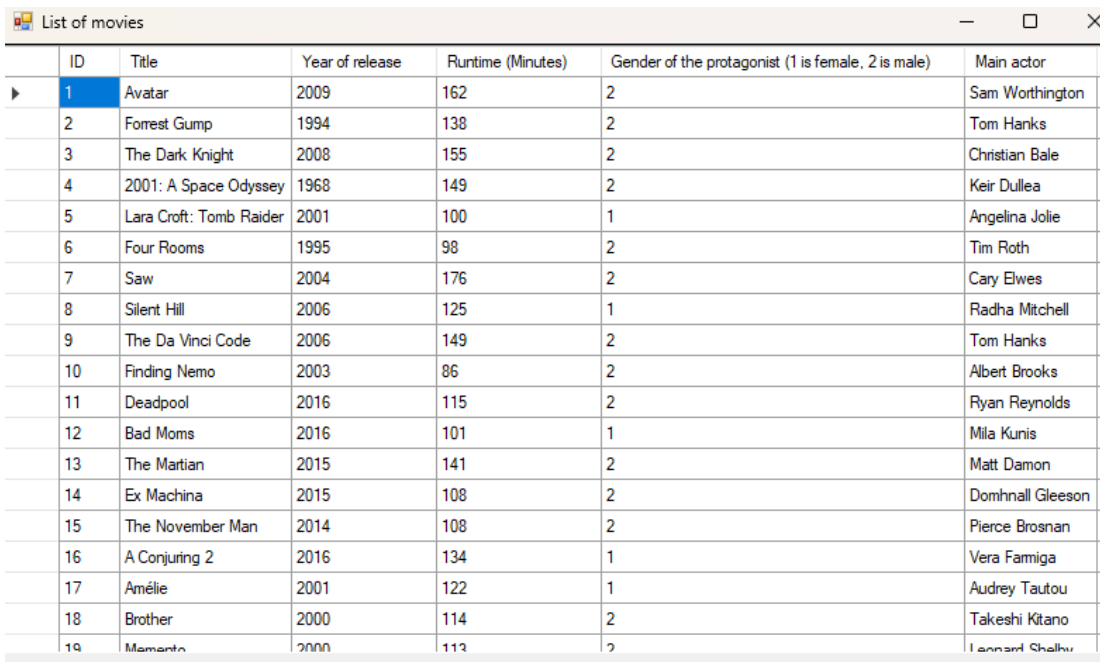
A programot lefuttatva a 18. ábrán látható képernyő fogadja a felhasználót, tehát ez gyakorlatilag a programnak a főmenüje, ahonnan indíthatóak a különböző funkciók az egyes gombokkal, amelyek menüpontokként szolgálnak.



18. ábra: A program főmenüje

#### 4.1.1 Az adatbázis tartalmának megjelenítése

A Table gomb segítségével a felhasználónak lehetősége van az adatbázis tartalmának megjelenítésére. A 19. ábrán látszik, hogy az adatok egy táblázat formájában jelennek meg, legfelül pedig láthatóak az oszlopok nevei, ez jelöli, hogy a filmek mely adatai vannak tárolva az adott oszlopban. A cellák csak olvashatóak, így nem lehet belekattintani, és ezzel átírni, módosítani az értéküket. A táblázat ablakmérete változtatható vagy akár teljes képernyőre is kinagyítható, ezzel még több adat fér el, és jeleníthető meg oldalra görgetés nélkül.

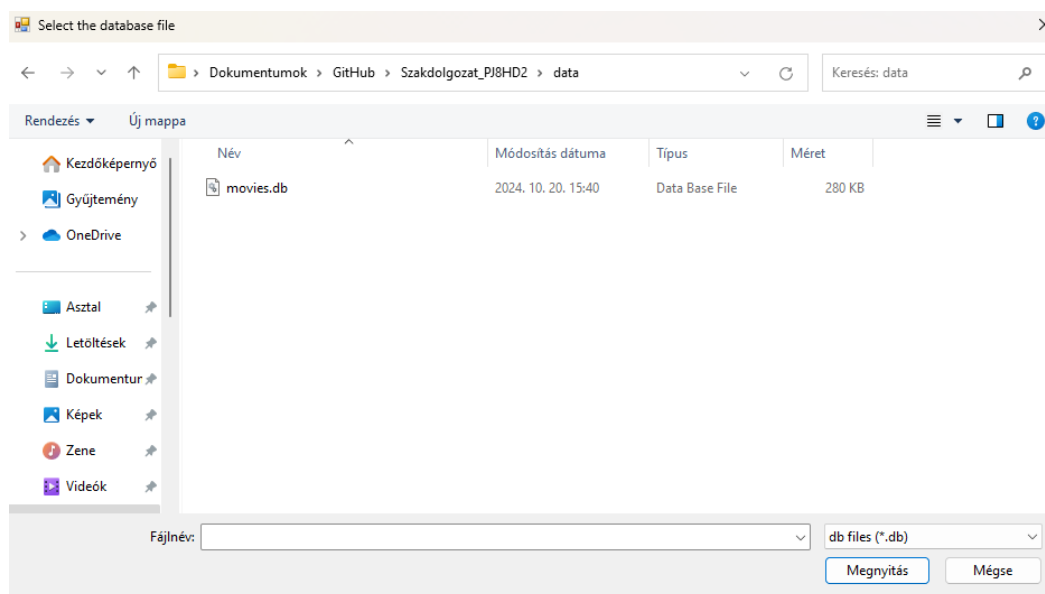


	ID	Title	Year of release	Runtime (Minutes)	Gender of the protagonist (1 is female, 2 is male)	Main actor
▶	1	Avatar	2009	162	2	Sam Worthington
	2	Forrest Gump	1994	138	2	Tom Hanks
	3	The Dark Knight	2008	155	2	Christian Bale
	4	2001: A Space Odyssey	1968	149	2	Keir Dullea
	5	Lara Croft: Tomb Raider	2001	100	1	Angelina Jolie
	6	Four Rooms	1995	98	2	Tim Roth
	7	Saw	2004	176	2	Cary Elwes
	8	Silent Hill	2006	125	1	Radha Mitchell
	9	The Da Vinci Code	2006	149	2	Tom Hanks
	10	Finding Nemo	2003	86	2	Albert Brooks
	11	Deadpool	2016	115	2	Ryan Reynolds
	12	Bad Moms	2016	101	1	Mila Kunis
	13	The Martian	2015	141	2	Matt Damon
	14	Ex Machina	2015	108	2	Domhnall Gleeson
	15	The November Man	2014	108	2	Pierce Brosnan
	16	A Conjuring 2	2016	134	1	Vera Farmiga
	17	Amélie	2001	122	1	Audrey Tautou
	18	Brother	2000	114	2	Takeshi Kitano
	19	Memento	2000	113	2	Leonard Shelby

19. ábra: Az adatbázis tartalmának, tehát a filmeknek a megjelenítése

## 4.1.2 Az adatbázis cseréje

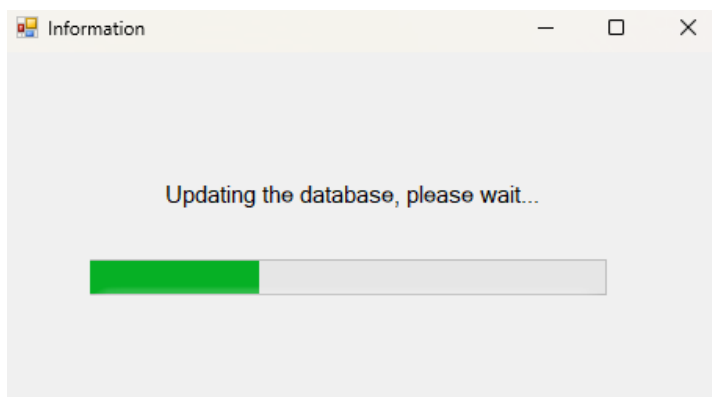
Ha a felhasználó le szeretné cserélni a program teljes adatbázisát, akkor azt a „Change database” gombbal teheti meg. Megnyomására megnyílik egy párbeszédablak, amelyben .db kiterjesztésű SQLite adatbázis fájlok betallózására nyílik lehetősége a felhasználónak (20. ábra).



20. ábra: A tallózáshoz használatos párbeszédablak

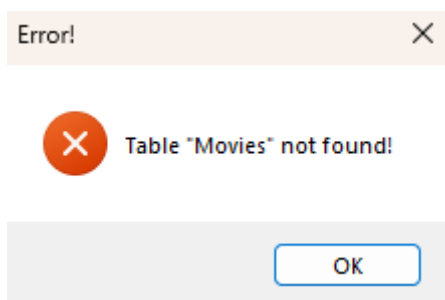
A kívánt fájlt betallózva amennyiben minden rendben van a fájlal, tartalmaz az eredeti movies.db adatbázisfájllhoz hasonlóan egy ugyanolyan szerkezetű *Movies* táblát, akkor a program lecseréli az eredeti adatbázisát a betallózottra, és megkezdzi a betallózott fájl

beolvasását, közben pedig megjelenik egy folyamatjelző csík, hogy tudassa a felhasználóval, a program dolgozik a háttérben (21. ábra).

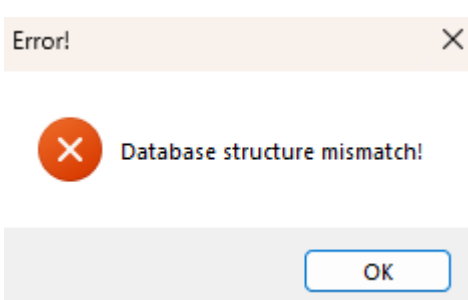


21. ábra: Folyamatjelző csík

Amennyiben a felhasználó véletlenül rossz adatbázis fájlt tállózott be, akkor arra a program felhívja a felhasználó figyelmét a megfelelő hibaüzenet felhasználásával (22. ábra, 23. ábra).



22. ábra: Hibaüzenet ha nem található a Movies tábla

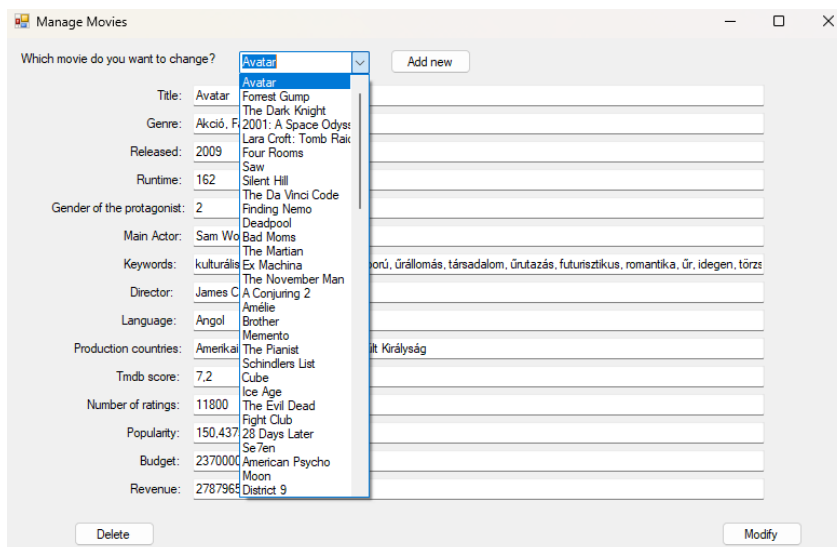


23. ábra: Hibaüzenet ha a Movies tábla struktúrája eltér az eredetitől

### 4.1.3 Filmek kezelése

Abban az esetben, ha a felhasználó a programon belül szeretne egy-egy filmet hozzáadni, esetleg annak adatait módosítani, vagy törölni az adatbázisból, akkor a „Manage Movies” menüpont kiválasztásával lehetősége nyílik erre. A megnyíló ablakban az adatbázis filmjei közül egy legördülő lista segítségével tud válogatni a felhasználó, a kiválasztott film adatai jelennek meg a szövegdobozokban (24. ábra).

Miután a felhasználó kiválasztotta a megfelelő filmet, eldöntheti hogy azt ki szeretné törölni, vagy valamely adatán módosítani szeretne. A „Delete” gomb megnyomása előtt nincs semmi teendő, a módosítás gomb megnyomása előtt viszont a szövegdobozokban szükséges módosítani azokat az adatokat, amelyeket meg szeretne változtatni a felhasználó, és ezután megnyomva hajtódnak végre a módosítások. Ameddig a program a háttérben dolgozik, a folyamatjelző csík itt is megjelenik (21. ábra).



24. ábra: Legördülő listából való választás

Az új film hozzáadására szolgáló „Add new” gomb megnyomásával a 24. ábrán a háttérben látható szövegdozok kiürülnek, és a felhasználó feltöltheti őket saját maga által meghatározott tetszőleges adatokkal. Mind a módosítás, mind pedig a hozzáadás esetén ha üresen maradnak szövegdozok, vagy nem megfelelő típusú, vagy nem megfelelő formájú adat van benne, akkor arra hibaüzenetek figyelmeztetik a felhasználót. Azoknál a mezőknél, ahol nem teljesen egyértelmű, hogy milyen adatnak kellene belekerülnie és milyen formában, ott az egeret a mező fölé mozgatva egy-egy címke nyújt segítséget a felhasználó számára (24. ábra).

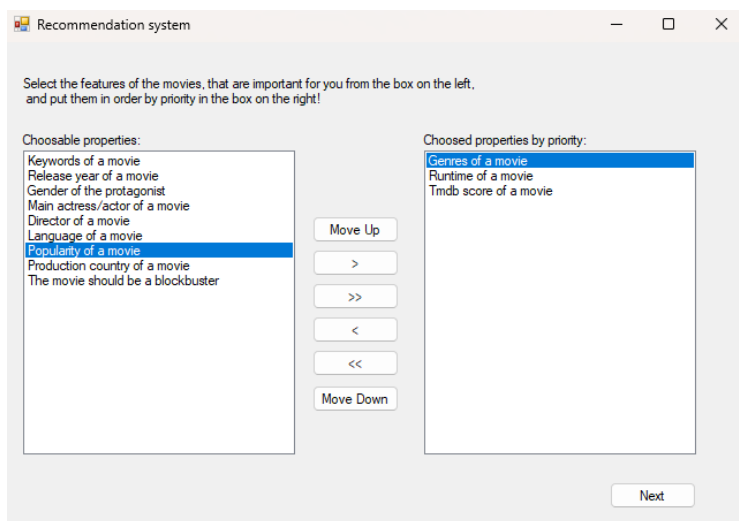
Runtime:	<input type="text" value="1 (male) or 2 (female)"/>
Gender of the protagonist:	<input type="text"/>
Main Actor:	<input type="text"/>

24. ábra: A főszereplő nemének beírásához segítő nyújtó címke

#### 4.1.4 Film ajánlása

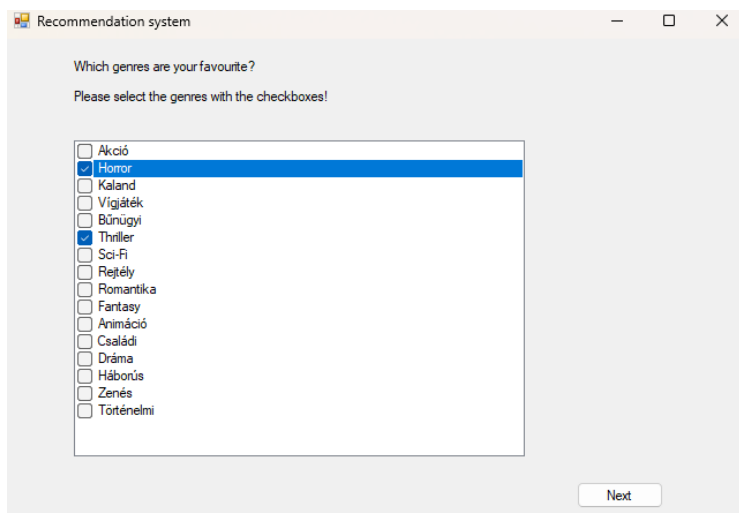
A főmenü „Recommendation System” menüpontját kiválasztva érhető el a program fő funkciója, az ajánlórendszer rész. A megnyitása után a felhasználót egy prioritásválasztó képernyő fogja fogadni, amelyen a képernyőn található gombok segítségével tudja a bal oldali listából kiválasztani, átmozgatni a filmeknek azokat a jellemzőit, tulajdonságait, amelyek számára fontosak, mikor filmeket keres, ezáltal csak azokra a jellemzőkre fog kérdéseket kapni, amelyek számára lényegesek (25. ábra). A jobb oldali listában, amelybe a kiválasztott tulajdonságok kerültek, nem csak az fontos, hogy milyen jellemzők kerültek bele, hanem az is, hogy azok milyen sorrendben vannak benne, erre pedig a program figyelmezteti is a felhasználót az ablak tetején található utasítás formájában. A lista beválogatott elemek prioritása tehát módosítható kijelölés (rákattintás) után az erre a célra létrehozott „Move up” és „Move down” gombokkal. Minél fentebb van egy jellemző a prioritási listában, annál nagyobb súllyal veszi figyelembe a program a rá adott választ, így annál jobban befolyásolja a

végeredményképpen adott ajánlott filmeket. Minél több jellemző kerül kiválasztásra, a program annál pontosabb ajánlásokat tud adni a felhasználó számára.



25. ábra: A prioritásválasztó képernyő

A prioritásválasztó képernyőről továbbhaladva a felhasználó megkapja azokat a kérdéseket, amelyek a kiválasztott jellemzőkhöz tartoznak. A 25. ábrán a filmek műfaja, a hossza, és a TMDb pontszáma került kiválasztásra. Ennek megfelelően az első kérdésben a műfajokat kell kiválasztanunk. Jelen esetben most „Horror” és „Thriller” műfajú filmeket szeretnénk nézni, így ennek megfelelően választunk (26. ábra).

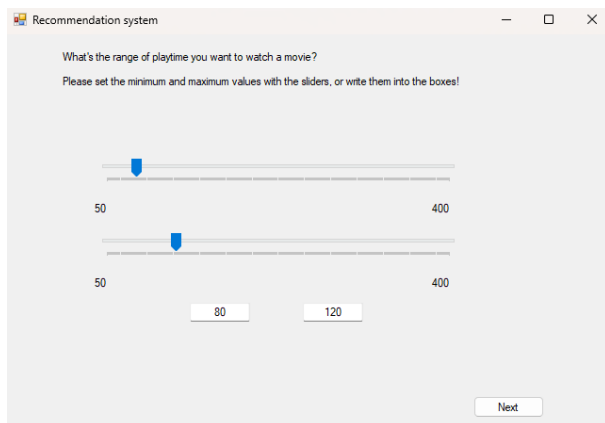


26. ábra: A műfajok kiválasztása

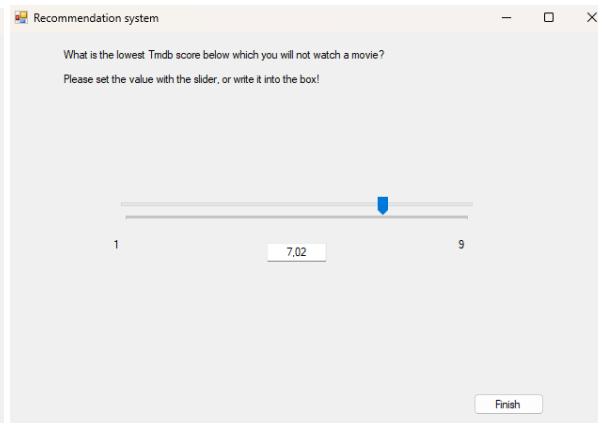
A 2. jellemző a film hosszára, a 3. pedig a TMDb pontszámára irányult, így ebben a sorrendben történt a megjelenítésük a műfajok kiválasztása után. A film hosszánál 2 csúszka segítségével lehet beállítani azt az időintervallumot, amelyen belül szeretnénk, hogy legyen az ajánlott film (28. ábra). Azonban a TMDb pontszámnál erre csak 1 csúszka áll rendelkezésünkre, mivel csak azt a minimum pontszámot tudjuk meghatározni, amelynél nem szeretnénk, hogy kevesebb legyen az ajánlott filmek pontszáma (29. ábra). A film hosszánál 80 és 120 perc közötti intervallum került



beállításra, a TMDB pontszámmal pedig 7,02 lett megadva, ennél nem szeretnénk alacsonyabb TMDB pontszámmal rendelkező filmeket kapni.

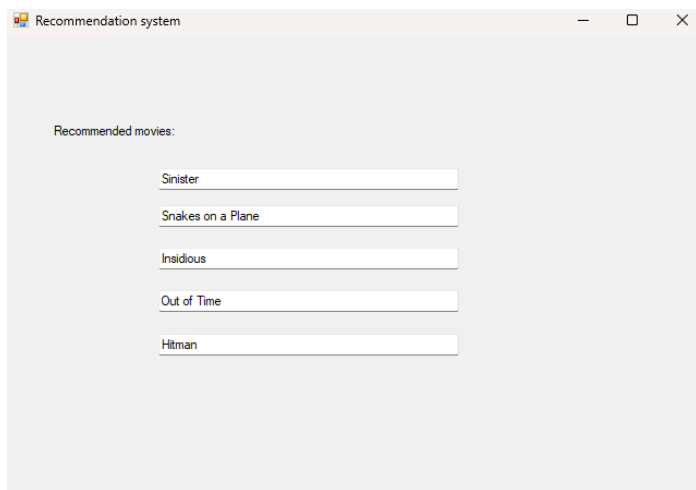
The screenshot shows a window titled "Recommendation system". The text inside asks: "What's the range of playtime you want to watch a movie? Please set the minimum and maximum values with the sliders, or write them into the boxes!". There are two horizontal sliders. The top slider has a range from 50 to 400, with a blue handle positioned at 80. The bottom slider also has a range from 50 to 400, with a blue handle positioned at 120. Below each slider is a text input box containing the value 80 and 120 respectively. A "Next" button is at the bottom right.

27. ábra: A film hosszának meghatározása

The screenshot shows a window titled "Recommendation system". The text inside asks: "What is the lowest Tmdb score below which you will not watch a movie? Please set the value with the slider, or write it into the box!". There is a horizontal slider with a range from 1 to 9, with a blue handle positioned at 7.02. Below the slider is a text input box containing the value 7.02. A "Finish" button is at the bottom right.

29. ábra: Az ajánlott filmekre vonatkozó minimum TMDB pontszám meghatározása

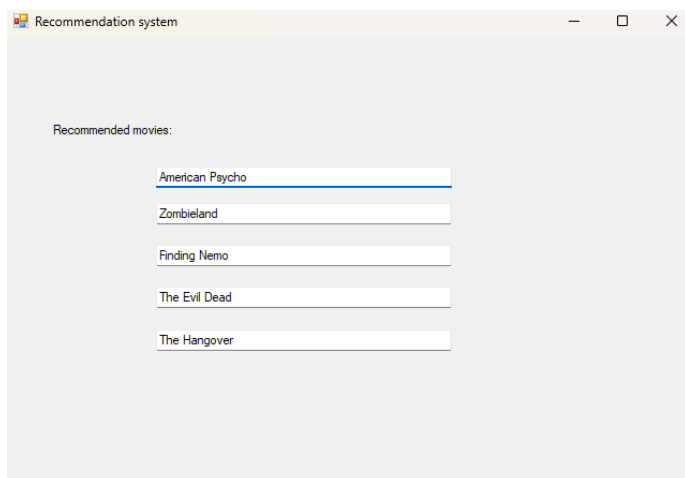
Miután a TMDB pontszámot is beállítottuk, és továbbléptünk, a program összeállítja az ajánlott filmek listáját (30. ábra). A listában legfelül lévő film az, amely a legpontosabban megfelel a felhasználó igényeinek. Esetünkben ez a „Sinister” című film, amelynek a műfaja „Horror, Thriller, Rejtély”, hossza 110 perc, TMDB pontszáma pedig 6,7. A mi esetünk Horror és Thriller műfajú filmeket szerettünk volna kapni minimum 80, de maximum 120 perces játékidővel, és legalább 7,02-es TMDB értékeléssel. A Sinister című film műfaja és hossza is megfelel a keresett jellemzőknek, a TMDB értékelése viszont elmarad a beállított pontszámtól. Ez nem meglepő, mivel a TMDB pontszám a kiválasztott jellemzők közül a prioritási listában a lista legalján volt, így a 3 válasz közül ezt vette figyelembe a legkevésbé a program az ajánlott filmek meghatározásánál.

The screenshot shows a window titled "Recommendation system". The text inside says: "Recommended movies:". Below this text is a list of five movie titles, each in a text input box: "Sinister", "Snakes on a Plane", "Insidious", "Out of Time", and "Hitman".

30. ábra: A megadott válaszok alapján az ajánlott filmek listája

Ha csinálunk egy próbát úgy, hogy ugyanezeket a jellemzőket válogatjuk be a prioritási listába, csak az előbbi példával pont fordított sorrendben (tehát TMDB pontszám, hossz,

műfajok lesz a sorrend) és a kérdésekre is az előbbieken bemutatott válaszokat adjuk, akkor eltérő eredményt kapunk (31. ábra).



31. ábra: Az ajánlott filmek a prioritási sorrend megfordításával

Ebben az esetben az „American Psycho” című film az, amely a válaszok alapján a legjobban megfelel a felhasználói igényeknek. Ennek a filmnek a TMDB értékelése 7,3, a játékidője 102 perc, a műfaja pedig Thriller, Dráma és Bűnügyi, tehát ebben az esetben a műfajai nem felelnek meg teljesen a beállított Horror és Thriller műfajokkal, mivel a Horror nem szerepel a film műfajai között. Azonban most a prioritási listában a műfajok voltak a lista legalján, így a 3 megadott jellemző közül ezt vette a legkevésbé figyelembe a program. Tehát innen látszik, hogy nem mindegy, hogy a listába milyen sorrendben kerülnek bele a jellemzők, a program figyelembe veszi azok prioritását az ajánlás elkészítéséhez.

## 4.2 Futtatható kód előállítás a forráskódból

A programomat a Visual Studio fejlesztőkörnyezetben írtam meg, amely a StackOverflow 2024-es felmérése szerint a mai napig az egyik legnépszerűbb fejlesztőkörnyezet [36]. A Visual Studioba be van építve a Roslyn fordító, amely valós, időben, a kód írásakor már képes jelezni, hogy ha valamilyen hiba, elírás történik a kódban. A Roslyn segítségével C#, és akár Visual Basic nyelven írt forráskódok is fordíthatóak futtatható kódokká [37].

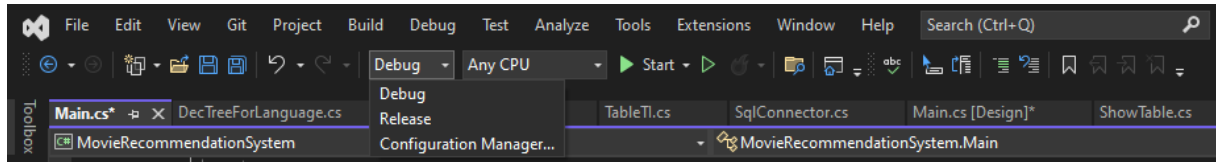
### 4.2.1 Fordítás Debug módban

Visual Studioban a fordítónak 2 külön konfigurációja van, a Debug és a Release. A Visual Studioba belépve alapértelmezett futtatási móda a „Debug”. Miután a programunk megírásra került, és a „Start” gomb megnyomásával a programunkat le szeretnénk futtatni, az „Debug”, tehát hibakereső módban fog lefordulni, amennyiben azt nem állítottuk át [38].

Debug módban a kód hibakeresés közben szerkeszthető a kód, az optimalizációk ki vannak kapcsolva, és futásidejű mappák használata történik. Az inicializálatlan változók memóiahelye 0xCC értékkel kerül feltöltésre [38].

## 4.2.2 Fordítás Release módban

A Visual Studio felső menüsorában a „Start” gomb mellett állítható át a futtatás módja.



18. ábra: A Visual Studio menüsora

Release módban futtatva a programot a Visual Studio figyelmeztet minket arra, hogy a programot „Release” módban futtatva a hibakeresés nem lesz olyan hatékony, de így is lehetőségünk van folytatni, vagy kilépni. Közben a program könyvtárában a Debug mappa mellett a fordítást követően létrejött programfájlok bekerülnek a korábbiakban még üres „Release” mappába, köztük az .EXE fájl is, amivel innentől a Visual Studio megnyitása nélkül is lehetőségünk nyílik bármikor lefuttatni a programot.

## 4.2.3 Debug és Release mód összehasonlítása

Release módban futtatva a programunkat érhetnek minket olyan váratlan hibák, amik Debug módban nem feltétlenül okoztak problémát. Például, ha egy változó Debug módban nincs inicializálva, akkor is rendben le kell futnia a programnak, mivel a változók memóiahelye 0xCC értékkel kerül feltöltésre. Ez Release módban nem így történik, mivel véletlenszerűen kerülnek kiosztásra a memóriacímek, így ez nem várt problémákat okozhat [38].

Ennek a fordítottja is igaz lehet, tehát Release módban elképzelhető hogy nem figyelmeztet minket a program olyan dolgokért, amikért Debug módban szólt, mivel a Debug mód, mint a nevéből is adódik leginkább hibakeresésre lett kitalálva, így sokkal több ellenőrzés hajtodik végre rajta, mint Release módban való futtatáskor, de ezáltal a futási teljesítmény is romlani fog, lassabb lesz, mint Release módban [38].

Mivel a Release módban fordított programkód során egy másik mappába kerülnek az összekészített programfájlok, mint előtte a fejlesztés során a Debug esetén, ezért ha az adott program tartalmaz olyan kódrészeket, amelyek fájlokat olvasnak be, akkor különös figyelmet igényel, hogy ezek az olvasandó fájlok a Release mappában is benne legyenek, mivel ameddig nincsenek benne, hibát fog okozni azok hiánya.

## 5. Összefoglalás

Szakdolgozatom célját sikerült megvalósítanom, tehát sikerült elkészítenem egy olyan ajánlórendszert filmekre vonatkozóan, amely a felhasználó igényeihez tud igazodni, és annak megfelelően feltenni a kérdéseket. Az ajánlórendszer futtatásának az elején megjelenő prioritási lista tartalmát a felhasználó tudja testreszabni, ennek a tartalmát használja fel a program arra a célra, hogy eldöntse, mely kérdések megjelenítésére van szükség, és milyen sorrendben az adott felhasználó esetében.

Az ajánlórendszerek bemutatása mellett részletezésre kerültek még más, egyéb logikán alapuló keresőalgoritmusok is, amelyek között voltak hagyományosabb koncepciókat használó keresők is, mint például a kulcsszó alapú keresők, de emellett voltak modernebb megközelítések is, mint például a kép alapú keresők, vagy az NLP (természetes nyelvfeldolgozás).

Az ajánlórendszerek témaköréhez kapcsolódó különböző témák, mint például a döntési fák, vagy az adatbázisrendszerek érintése mellett az ML.NET gépi tanulási algoritmusokat tartalmazó függvénykönyvtár is sikerült bemutatnom egy olyan szemszögből, hogy milyen becslésekkel kapcsolatos részeket, hatékonyságmérési algoritmusokat tartalmaz, továbbá hogy hogyan lehet a függvényeinek használatával becselő algoritmusokat készíteni. A függvénykönyvtár algoritmusainak segítségével elkészített különböző becselő algoritmusok hatékonyságának tesztelésével kiderült, hogy egy konkrét esetre hogyan működnek azok. Összevetésük során megvizsgálásra került, hogy folytonos értékek megbecslésére vonatkozó következtetések esetében az aktuális körülmények között mely algoritmusok tudtak hatékonyan működni, és mely algoritmusok működtek még a véletlen találgatástól is rosszabbul.

Az elkészített programból az a következtetés vonható le, hogy a C# programozási nyelv egy megfelelő alapot tud szolgáltatni egy olyan grafikus felületű, asztali alkalmazás létrehozásához, amely egy ajánlórendszert valósít meg. A becslések megvalósításához nagy segítséget tudott nyújtani az ML.NET függvénykönyvtár, mivel a benne megtalálható, már előre implementált algoritmusokat használva nincs szükség arra, hogy azokat a programozónak saját magának kelljen implementálnia, ezáltal a függvénykönyvtár használata lényegesen meg tudja könnyíteni a dolgát.

A programban használt két egyéb technológia, az SQLite és a GraphViz is egyaránt problémamentesen összekapcsolható volt, illetve együtt tudott működni a C# alkalmazással.

Szakdolgozatomban törekedtem arra, hogy bemutassam a programomat olyan szempontból, amely a programban használt logikákra, megoldásokra fókuszál implementációközelibb módon, illetve egy olyan, másik szempontból is, amely az elkészített programnak a használatát, működését mutatja be.

## 6. Summary

I have successfully fulfilled the main objective of my thesis, which was to design and develop a customizable recommendation system for movies. This system is able to dynamically adapt to the specific needs and preferences of each user with the priority list, prompting relevant questions accordingly. At the beginning of each recommendation, users are presented with a priority list that they can tailor to reflect their unique preferences about movies.

In addition to the introduction of the recommendation systems, my thesis also explores various other search algorithms that employ different methodologies. These include both traditional approaches, such as keyword-based search engines that filter results based on input keywords, and modern, innovative methods like image-based search engines or NLP (natural language processing).

Beyond covering various topics related to recommendation systems, such as decision trees and database systems, I also explored the ML.NET machine learning library, focusing on its estimation components, efficiency measurement tools, and how to construct estimation algorithms using its functions. By testing the efficiency of different estimation algorithms built with the library, I gained insights into their performance in specific cases. The algorithms were compared to assess which could efficiently perform continuous value estimation under real conditions and which performed worse than even random guessing.

In conclusion, C# provides an excellent foundation for creating a desktop application with a graphical user interface that implements a recommendation system. The ML.NET library is highly beneficial for predictive tasks, as its pre-implemented algorithms can save so much time for the programmer, since it's not necessary for the programmer to implement them manually, which can take a lot of time, as it can be a long and complicated process. It's great feature to simplify the development process.

The two other technologies used in the program, SQLite and GraphViz, were easy to integrate and worked seamlessly with the C# application.

In writing my thesis, I sought to present a dual perspective on the application. On one hand, I provided an implementation-oriented view that focuses on the logical approaches and solutions utilized in the program's development. On the other hand, I aimed to highlight the practical usage and functionality of the system from the user's perspective. This two-pronged approach allowed me to communicate both the technical complexities of the recommendation system and its usage as a functional tool.

## 7. Köszönetnyilvánítás

Ezúton szeretném kifejezni hálámat mindazoknak, akik szakdolgozatom elkészítése során támogattak és segítettek. Külön köszönet illeti Glavosits Tamás Tanár Urat, aki vállalta a témavezetői szerepet, és végigkísérte munkám folyamatát. Értékes visszajelzései és iránymutatásai nagy segítséget jelentettek számomra a dolgozat elkészítésében, hozzájárulva a munka magas szakmai színvonalához. Hálás vagyok a Tanár Úrnak, amiért támogatta szakdolgozatom létrejöttét, és értékelésével segített a dolgozat formálásában.

Továbbá külön köszönet illeti Piller Imre Tanár Urat, aki aktív közreműködésével és értékes szakmai meglátásaival segített a szakdolgozatom kivitelezésében. Segítőkészsége és értékes tanácsai végigkísértek a folyamat során, és sokat jelentettek számomra, mind a dolgozat elkészítésében, mind a szakmai fejlődésemben.

Mindkét Tanár Úrnak köszönöm az iránymutatást és a bizalmat, amelyek ösztönzőleg hatottak rám a dolgozat elkészítése során.

## 8. Irodalomjegyzék

- [1]: Szegedi Tudományegyetem Informatika Intézete: Reprezentáció tanulás, Letöltés dátuma: 2024. 10.11. <https://www.inf.u-szeged.hu/~rfarkas/ML22/embedding.html>
- [2]: Aminu Da'u & Naomie Salim: Recommendation system based on deep learning methods: a systematic review and new directions, Springer Nature, 2019: <https://link.springer.com/article/10.1007/s10462-019-09744-1>
- [3]: Mohammad Aamir, Mamta Bhusry: Recommendation System: State of the Art Approach, International Journal of Computer Applications, 2015: <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=199700bb1e2329b973c96b019d8c7d6910e6911a>
- [4]: Sayali D. Jadhav, H. P. Channe: Efficient Recommendation System Using Decision Tree Classifier and Collaborative Filtering, International Research Journal of Engineering and Technology, 2016: <https://www.irjet.net/archives/V3/i8/IRJET-V3I8393.pdf>
- [5]: J. R. Quinlan: Learning decision tree classifiers, Association for Computing Machinery, 1996: <https://dl.acm.org/doi/pdf/10.1145/234313.234346>
- [6]: Jiang Su, Harry Zhang: A Fast Decision Tree Learning Algorithm, Association for the Advancement of Artificial Intelligence, 2006: <https://cdn.aaai.org/AAAI/2006/AAAI06-080.pdf>
- [7]: SQLite hivatalos weboldala, Letöltés dátuma: 2024. 10.11.: <https://www.sqlite.org/>
- [8]: <https://www.kaggle.com/datasets/TMDB/TMDB-movie-metadata>
- [9]: TMDB hivatalos weboldala, Letöltés dátuma: 2024. 10.20.: <https://www.themoviedb.org/>
- [10]: Ankita Malve, Prof. P. M. Chawan: A Comparative Study of Keyword and

Semantic based Search Engine, International Journal of Innovative Research in Science, Engineering and Technology, 2015:

[https://www.researchgate.net/profile/Pramila-Chawan/publication/316514673\\_A\\_Comparative\\_Study\\_of\\_Keyword\\_and\\_Semantic\\_based\\_Search\\_Engine/links/5901cfc7a6fdcc8ed51110da/A-Comparative-Study-of-Keyword-and-Semantic-based-Search-Engine.pdf](https://www.researchgate.net/profile/Pramila-Chawan/publication/316514673_A_Comparative_Study_of_Keyword_and_Semantic_based_Search_Engine/links/5901cfc7a6fdcc8ed51110da/A-Comparative-Study-of-Keyword-and-Semantic-based-Search-Engine.pdf)

[11]: Martin Carlstedt: Using NLP and context for improved search result in specialized search engines, Mälardalen University, 2017:

<https://www.diva-portal.org/smash/get/diva2:1088795/FULLTEXT01.pdf>

[12]: Shaidah Jusoh: A study on NLP applications and ambiguity problems, Journal of Theoretical and Applied Information Technology, 2018:

<https://www.jatit.org/volumes/Vol96No6/4Vol96No6.pdf>

[13]: Marcelo Arenas, Bernardo Cuenca Grau, Evgeny Kharlamov, Sarunas Marciuska, Dimitriy Zheleznyakov: Faceted search over RDF-based knowledge graphs, Journal of Web Semantics, 2016:

<https://www.sciencedirect.com/science/article/abs/pii/S1570826815001432>

[14]: Osmo Suominen, Kim Viljanen, E. Hyvönen: User-Centric Faceted Search for Semantic Portals, Extended Semantic Web Conference, 2007:

[https://link.springer.com/chapter/10.1007/978-3-540-72667-8\\_26](https://link.springer.com/chapter/10.1007/978-3-540-72667-8_26)

[15]: K.Kranthi Kumar, Dr.T.Venu Gopal: CBIR: Content Based Image Retrieval, National Conference on Recent Trends in information/Network Security, 2010:

[https://www.researchgate.net/profile/K-Kranthi-Kumar-2/publication/235634738\\_CBIR\\_Content\\_Based\\_Image\\_Retrieval/links/004635189dd9ac1e6f000000/CBIR-Content-Based-Image-Retrieval.pdf](https://www.researchgate.net/profile/K-Kranthi-Kumar-2/publication/235634738_CBIR_Content_Based_Image_Retrieval/links/004635189dd9ac1e6f000000/CBIR-Content-Based-Image-Retrieval.pdf)

[16]: SQLite hivatalos weboldalának kezdőlapja, Letöltés dátuma: 2024. 10.11.:

<https://www.sqlite.org/index.html>

[17]: SQLite hivatalos weboldalának dokumentációja, Letöltés dátuma: 2024. 10.11.:

<https://www.sqlite.org/whentouse.html>

[18]: StackOverflow felmérése, Letöltés dátuma: 2024. 10.11.:

<https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database>

[19]: System.Data.SQLite csomag hivatalos weboldala, Letöltés dátuma: 2024. 10.11.:

<https://system.data.sqlite.org/index.html/doc/trunk/www/index.wiki>

[20]: Adnan Kattakaden: Exploring the Evolution, Benefits, and Limitations of JSON: A Beginner's Guide, Medium cikke, 2023, Letöltés dátuma: 2024. 10.11.:

<https://medium.com/@adnankattakaden/exploring-the-evolution-benefits-and-limitations-of-json-a-beginners-guide-7d8ba5e1c5e4>

[21]: Google Cloud hivatalos dokumentációja, Letöltés dátuma: 2024. 08.18.:

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

[22]: Mlpack hivatalos weboldala, Letöltés dátuma: 2024. 10.11.:

<https://www.mlpack.org/>



- [23]: Scikit-learn hivatalos weboldala, Letöltés dátuma: 2024. 10.11.: <https://scikit-learn.org/stable/>
- [24]: Microsoft hivatalos dokumentációja az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: <https://dotnet.microsoft.com/en-us/apps/machinelearning-ai/ml-dotnet>
- [25]: ML.NET wikipédia oldala, Letöltés dátuma: 2024. 08.18.: <https://en.wikipedia.org/wiki/ML.NET>
- [26]: Microsoft hivatalos referencialeírása az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: <https://learn.microsoft.com/hu-hu/dotnet/machine-learning/how-does-mldotnet-work>
- [27]: Microsoft hivatalos dokumentációja az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: <https://dotnet.microsoft.com/en-us/learn/ml-dotnet/what-is-mldotnet>[36]
- [28]: Microsoft hivatalos referencialeírása az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: [https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/transforms?WT.mc\\_id=dotnet-35129-website](https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/transforms?WT.mc_id=dotnet-35129-website)
- [29]: Microsoft hivatalos referencialeírása az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: [https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/tasks?WT.mc\\_id=dotnet-35129-website](https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/tasks?WT.mc_id=dotnet-35129-website)
- [30]: Jonathan L. Herlocker, Joseph A. Konstan, John Riedl: Explaining collaborative filtering recommendations, Association for Computing Machinery, 2000: <https://dl.acm.org/doi/abs/10.1145/358916.358995>
- [31]: Alauddin Yousif Al-Omary, Mohammad Shahid Jamil: A new approach of clustering based machine-learning algorithm, ScienceDirect, 2006 <https://www.sciencedirect.com/science/article/abs/pii/S0950705106000189>
- [32]: Microsoft hivatalos referencialeírása az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: [https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/metrics?WT.mc\\_id=dotnet-35129-website](https://learn.microsoft.com/en-us/dotnet/machine-learning/resources/metrics?WT.mc_id=dotnet-35129-website)
- [33]: Ajitesh Kumar: Micro-average, Macro-average, Weighting: Precision, Recall, F1-Score, Analytics Yogi, Letöltés dátuma: 2024. 08.18.: [https://vitalflux.com/micro-average-macro-average-scoring-metrics-multi-class-classification-python/#What\\_Why\\_of\\_Micro\\_Macro-averaging\\_and\\_Weighting\\_metrics](https://vitalflux.com/micro-average-macro-average-scoring-metrics-multi-class-classification-python/#What_Why_of_Micro_Macro-averaging_and_Weighting_metrics)
- [34]: Microsoft hivatalos referencialeírása az ML.NET-ről, Letöltés dátuma: 2024. 08.18.: [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](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)
- [35]: Microsoft hivatalos referencialeírása az SDCA-ról, Letöltés dátuma: 2024. 10.30.: <https://learn.microsoft.com/en-us/dotnet/api/microsoft.ml.standardtrainerscatalog.sdca?view=ml-dotnet>
- [36]: StackOverflow felmérése, Letöltés dátuma: 2024. 10.30.: <https://survey.stackoverflow.co/2024/technology#most-popular-technologies-new-collab-tools>



- [37]: Microsoft hivatalos referencialeírása az Roslyn-ról, Letöltés dátuma: 2024. 10.30.: <https://learn.microsoft.com/en-us/visualstudio/extensibility/dotnet-compiler-platform-roslyn-extensibility?view=vs-2022>
- [38]: [Visual Studio] — Differences between Debug and Release, Medium cikke, 2020, Letöltés dátuma: 2024. 10.30.: <https://hanzli.medium.com/visual-studio-differences-between-debug-and-release-f9ebf3aa7a01>

## Melléklet

Ez a mellékletként szolgáló dokumentum részletesen bemutatja az elkészített ajánlórendszerem részeként szolgáló programok felépítését, kódszintű bemutatását. Az ajánlórendszert a TableInserts és a MovieRecommendationSystem programok alkotják, ezek közül a TableInserts program gyakorlatilag egy segédprogramként funkcionál a MovieRecommendationSystem számára, ezáltal grafikus felülettel nem is rendelkezik. A MovieRecommendationSystem szolgál a szakdolgozatom fő programjaként, ami már egy grafikus felülettel rendelkező C# alkalmazás, amely függvénykönyvtár formájában felhasználja a TableInserts funkcióit.

### A. A TableInserts program felépítése

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 felsorolásszerűen 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 azonosítót 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 azonosítójához hozzárendeljük annak az elemnek az azonosítóját, 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 *Movies* 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.

## A.1 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 *Movie* osztály adattagjai.

A *PrepareKeywords*, *PrepareLanguages*, *PrepareDirectors*, *PrepareCountries*, és *PrepareGenres* metódusok a film adott jellemzőit kigyűjtik egy listába, ahol minden egyes elem, ami az adatbázis *Movies* 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”.

## A.2 Program 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.

## A.3 Table osztály

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

Az 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* 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:

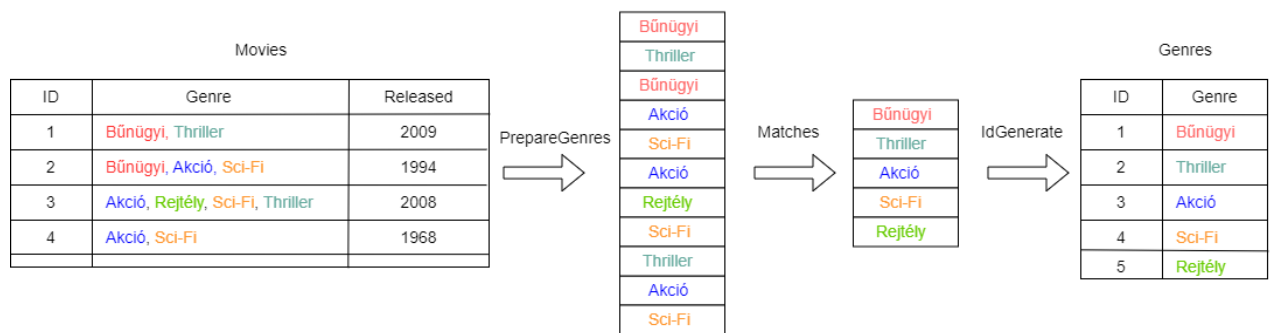
- táblát tölt fel: *Genre*, *Keyword*, *Language*, *Director*, *Country*,
- kapcsolótáblát tölt fel: *Movies\_Genres*, *Movie\_Keywords*, *Movie\_Languages*, *Movies\_Directors*, *Movies\_Countries*.

### A.3.1 Táblát feltöltő metódusok

Amelyek táblát töltenek fel, azok 6 darab metódushívást tartalmaznak. Az alábbiakban a *Genre* metódus tartalma kerül bemutatásra, de a *Keyword*, *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 *Genre* esetén a *PrepareGenres*. 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 műfajt, ömlesztve.
2. Az *Algorithms* osztály *Matches* metódusának meghívása.  
A metódusnak átadjuk az előzőleg feltöltött *data* listát, illetve létrehozunk benne egy átmeneti string típusú listát (*filteredGenres*). Végigmegyünk a *data* minden elemén, és ami még nem szerepel a *filteredGenres* -ben, azt hozzáadjuk, ha pedig már szerepel benne az adott elem akkor megyünk a következőre. A metódus visszatérési értéke egy string típusú lista, így a *filteredGenres*-t fogjuk belőle visszaadni. A metódushívás helyén a *data* listát egyenlővé tesszük magával a metódushívással, tehát gyakorlatilag a *filteredGenres* lista tartalmával.
3. 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.
4. Az adott táblához kapcsolódó *FillUp* metódus meghívása a *FillUps* osztályból.  
A műfajok esetén *FillUps.Genres*. Ezzel a metódussal történik az adatbázis adott táblájának feltöltéséhez szükséges utasítások összeállítása, jelen esetben a *Genre* 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.
5. 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.
6. 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.

Hogy a *Genre* tábla feltöltéséhez szükséges különböző előkészítő szakaszokban, metódushívások után milyen állapotban vannak a műfajok, azt az 1. ábra szemlélteti.



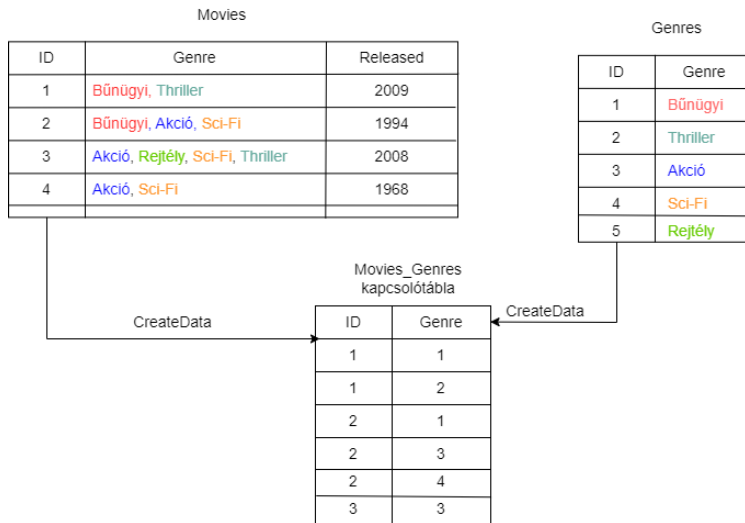
8. ábra: A Genre tábla feltöltének előkészítése

### A.3.2 Kapcsolótáblát feltöltő metódusok

Azok a metódusok, amelyek kapcsolótáblát töltenek fel, azok tartalmazzák azokat a metódushívásokat is, amelyek az előbbieken, a sima tábláknál kerültek bemutatásra. Az alábbiakban a *Genre\_Movie* metódus tartalma kerül részletezésre, de a *Keyword\_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.
2. *PrepareGenres* meghívása
3. Az *Algorithms* osztály *Matches* metódus hívása
4. Az *Algorithms* osztály *IdGenerate* meghívása
5. Az *Algorithms* osztály *CreateData* függvény meghívása
6. *FillUp\_MoviesKeywords* meghívása
7. *ExecuteInserts* metódussal a parancsok végrehajtása
8. *cmds.Clear()* használatával a lista kiürítése

Hogy a kapcsolótáblák mely adataikat melyik táblából kapják, azt a 2. ábra szemlélteti.



9. ábra: A kapcsolótáblák felépítése

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

A *CopyForCreateData* 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 műfajokat 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 műfajnak az azonosítóját az *id* listából, amellyel az egyezés történt. Így létrejönnek a kapcsolótábla feltöltéséhez szükséges adatok.

## B. MovieRecommendationSystem program felépítése

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 az ajánlás elindítása után kérdéseket tesz fel a felhasználónak, amelyek megválaszolása után tudja meghatározni az alkalmazás az ajánlott filmeket. A program szintén a *movies.db* adatbázisból dolgozik, így ezt használja az ajánlott film kiválasztásához is.

## B.1 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 tároló 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 következők 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 segítségével beazonosítjuk a műfajok neveit,
- *GenreStringWithCommas*: A műfajok neveinek vesszővel elválasztott felsorolása táblázatos megjelenítéshez.

## B.2 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 segédprogramként 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, törölünk, 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.

### B.3 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 *SqlConnector*-ban található különböző *Get* metódusokban 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.

### B.4 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 *Movies* 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 egész értékek 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 azonosítókön 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 azonosítóit és neveit kérdezzük le az adatbázisnak az aktuális jellemzőjéhez tartozó táblából, jelen esetben a műfajok esetében a *Genres* táblából, és ezen adatokat paraméterként átadva meghívjuk az *Algorithms* osztály 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 integerok 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);
}

```

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.

Az osztály tartalmaz egy további metódust, a *GetDbStructure*-t, amelyet a későbbiekben a *Main* osztályban fog a program felhasználni abban az esetben, ha a felhasználó le szeretné cserélni a program adatbázisát egy másikra. Ez a metódus paraméterként egy SQLite adatbázis elérési útját kapja meg, és ennek az adatbázisnak a *Movies* táblájának visszaadja a struktúráját egy szöveges változó formájában. Először létrehozunk egy string típusú *structure* változót üres értékkel, majd a paraméterként kapott fájlal létrehozunk egy SQL kapcsolatot. Az SQLite beépített *sqlite\_master* táblája tárolja az adott adatbázis szerkezetét táblafeltöltő parancsok formájában. Ebből egy SQL parancs segítségével kiolvassuk azt a táblafeltöltő parancsot, amely a *Movies* táblához tartozik. Amennyiben az adatbázis nem rendelkezik *Movies* táblával, akkor a program hibát fog dobni, amit egy kivételkezeléssel lekezelünk, majd a program egy felugró ablak formájában a *Movies* tábla hiányát a felhasználó tudtára adja. Ez esetben a metódus egy üres szöveges változót fog visszaadni.

## B.5 Algorithms osztály

A *Movie* típusú *movies* lista azon adattagjait, amelyek nem skalár értékek, 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. Inicializálásuk 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 filmekben 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 segítségével elvégzik a jellemzők neveinek is a hozzárendelését.

A filmekben végigmegyünk, majd pl. a műfajok esetén 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 *TableInserts*nek 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.

## B.6 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 fájlnek az elkészítése történik a fa feldolgozása közben.

A *MainJson* metódusban a fát először beolvassuk a json forrásfájlból, majd egy *JsonObject*-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 JsonObject* 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 elágazásai, illetve azoknak a további alsó (közvetett) elágazásai bekerülnek a *JArray* típusú *conditionArray* tömbbe. Tehát a metódus első futáskor 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 csomópontnak 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  $\leq 6,0$ ) esetén. Továbbá az újonnan hozzáadott csomóponthoz címkéként 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.

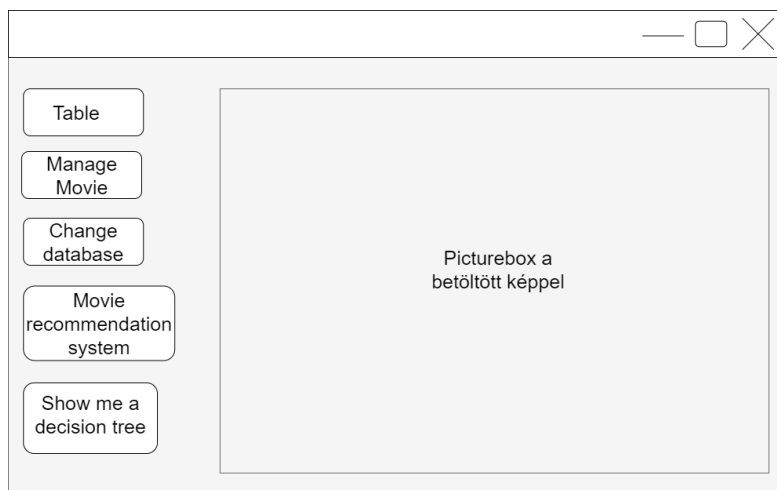
## B.7 ProgressBar osztály

A *ProgressBar* is egy grafikus ablakot megjelenítő osztály, amelynek a megjelenítésére az adatbázis frissítésekor kerül sor. Az osztály a konstruktoron kívül nem tartalmaz egy metódust, 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.

## B.8 Main osztály

A program fő osztálya a *Main*, ami a program fő ablakát, főmenüjét jeleníti meg, ennek képernyőterve a 3. ábrán látható.



10. ábra: A Main osztály, tehát a főmenü képernyőterve

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, és a főképernyőn megjelenő kép betöltése és megjelenítése is.

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

A főmenü tartalmaz egy „Table”, egy „Change database”, egy „Manage Movie”, egy „Recommendation System”, és egy „Show me a decision tree” 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. Ezek közül a „Show me a decision tree” gomb egy egyszerű metódushívás, ami megjeleníti a kézzel készített döntési fának a kirajzolását, ami a korábbiakban már részletezésre került, viszont a többi gomb új funkciókat valósít meg a programban. 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 törölünk 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 a metódusokat is, amelyeket a *Main\_Load*-ban a program indulásakor tettünk.

### B.8.1 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. A metódus további részeiben beállításra kerülnek a táblázat egyes mezőinek a nevei, illetve az automatikus oszlopszélesség annak érdekében, hogy olyan szélesek legyenek, amiben a leghosszabb adat is elfér.

### B.8.2 Change Database gomb

A Change Database gomb megnyomásával a *ChangeDB\_Click* metódus fut le. Lefutásakor a Windows Intéző tallózó ablaka nyílik meg, amely segítségével betallózzhatunk egy másik, viszont fontos, hogy csak a *movies.db* szerkezetével teljes mértékben megegyező SQLite adatbázis fájlt. A betallózással megkapjuk a betallózandó fájl elérési útját.

A program az eredeti adatbázis *Movies* táblájának a szerkezetét, és a betallózandó adatbázisnak, amennyiben van *Movies* táblája, akkor annak is eltárolja a szerkezetét. Ezt az *SqlConnector* osztályban korábban már említett *GetDbStructure* metódussal tudja megtenni a program. A kapott struktúrákat egy-egy szöveges változóban tároljuk el. A *ChangeDB\_Click* metódusban ez a két változó összehasonlításra kerül, és csak abban az esetben kerül sor az adatbázis cseréjére, ha a két szöveges érték megegyezik, tehát a két adatbázis *Movies* tábláinak struktúrája megegyezik. Amennyiben a betallózott adatbázis nem rendelkezik *Movies* táblával, akkor a betallózandó adatbázis struktúráját tároló szöveges érték üresen marad, tehát ebben az esetben sem tud megtörténni a csere.

Amennyiben a struktúrák egyeznek, akkor a betallózott fájlt átmásoljuk a program saját könyvtárába *movies.db* fájlnevvvel, 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.

### B.8.3 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.

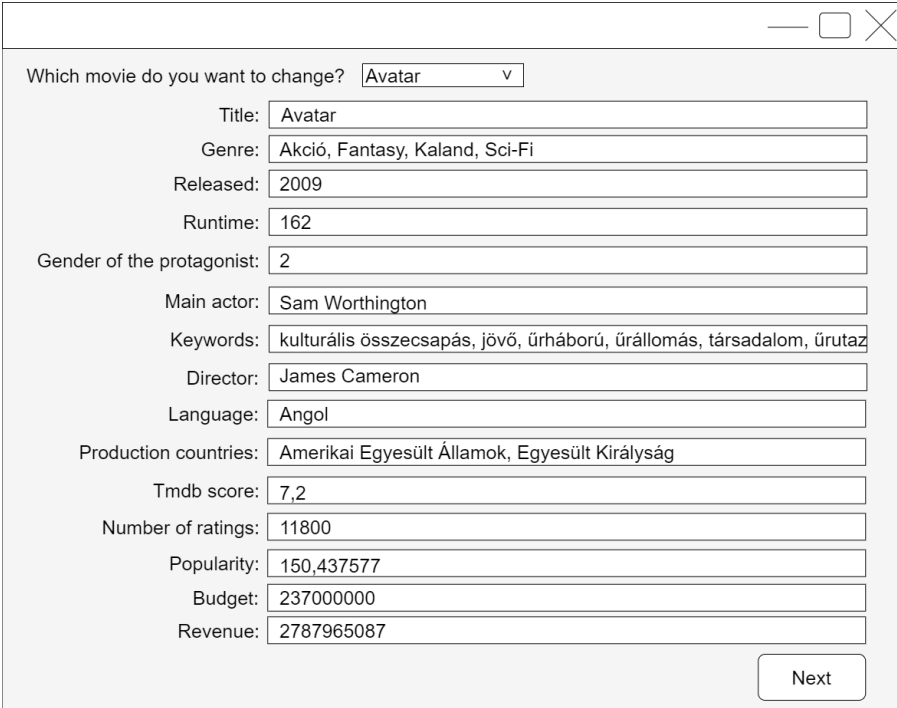
A *ManageMovie* osztály segítségével kezelhetjük az adatbázisunkat a programon belül. Hozzáadhatunk új filmeket, törölhetünk belőle, vagy akár módosíthatjuk is azoknak a különböző tulajdonságait.

A megjelenített ablak képernyőtervét a 4. ábra mutatja be. 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 legördülő 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.



Which movie do you want to change? Avatar v

Title:	Avatar
Genre:	Akcio, Fantasy, Kaland, Sci-Fi
Released:	2009
Runtime:	162
Gender of the protagonist:	2
Main actor:	Sam Worthington
Keywords:	kulturális összecsapás, jövő, úrháború, úrállomás, társadalom, űrutaz
Director:	James Cameron
Language:	Angol
Production countries:	Amerikai Egyesült Államok, Egyesült Királyság
Tmdb score:	7,2
Number of ratings:	11800
Popularity:	150,437577
Budget:	237000000
Revenue:	2787965087

Next

11. ábra: A *Manage Movie* gomb által megjelenített ablak képernyőterve

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 comboboxban, 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 kisebb 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 elrejt 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 *IsEmpty* 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 igénylő 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;
}

```

## B.9 Becslésekre vonatkozó osztályok

A programom fő célja ugyan az, hogy a felhasználónak az általa megválasztott 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.

### B.9.1 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.

### B.9.2 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.

### B.9.3 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 ismertette 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));

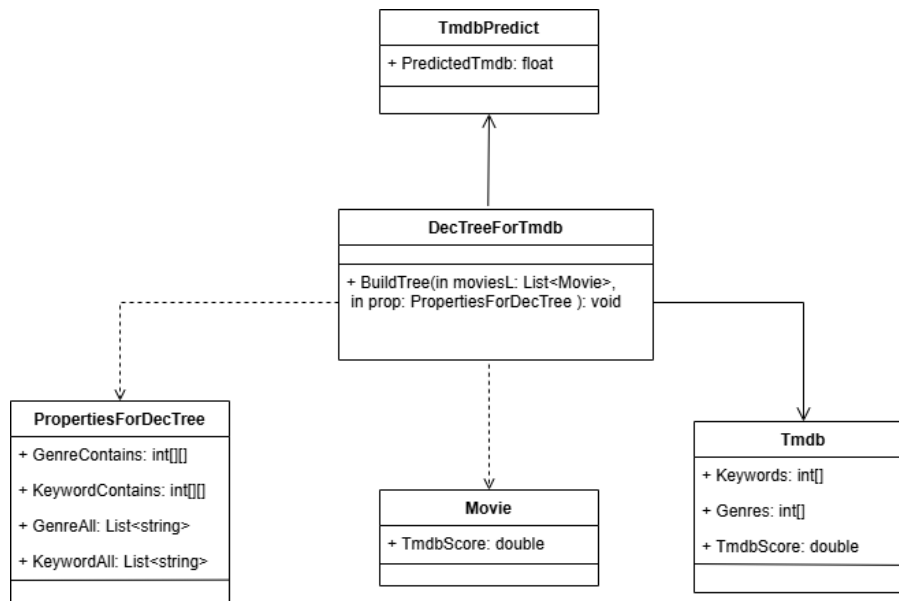
Console.WriteLine($"R^2: {metrics.RSquared}");
Console.WriteLine($"Mean Absolute Error: {metrics.MeanAbsoluteError}");
Console.WriteLine($"Mean Squared Error: {metrics.MeanSquaredError}");

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 pontszámra vonatkozó becslés osztálydiagramja (5. ábra) könnyebben áttekinthetővé teszi, hogy milyen osztályok, és azoknak mely adattagjai, metódusai vesznek részt a becslés elvégzésében.





12. ábra: Osztálydiagram a TMDB pontszám becslésére vonatkozóan

## B.10 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* algoritmus 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 következők: *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* változó, 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.