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 felületettel 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.

1. 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 felsorolás szerűen többel is), akkor arra rendkívül kicsi esély van, hogy az adott tulajdonságot vizsgálva a 2 film 100%-osan megegyezzen. Viszont ettől még lehet számos egyezés, azonban ha az adatokat egyben vizsgáljuk akkor ezeket nem tudjuk megtalálni, ezért szükséges ezen mezők lebontása, hogy a tartalmukban szereplő különböző adatokat külön-külön tudjuk kezelni, ne pedig egyben.

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

A lebontás folyamatának általános lépései:

1. Az adott lebontandó mező összes adatának kigyűjtése egy listába
2. A listából az ismétlések kiszűrése
3. A lista maradék elemeihez egyedi 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.

1.1 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”.

1.2 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.

1.3 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.*

1.3.1 Táblát tölt fel

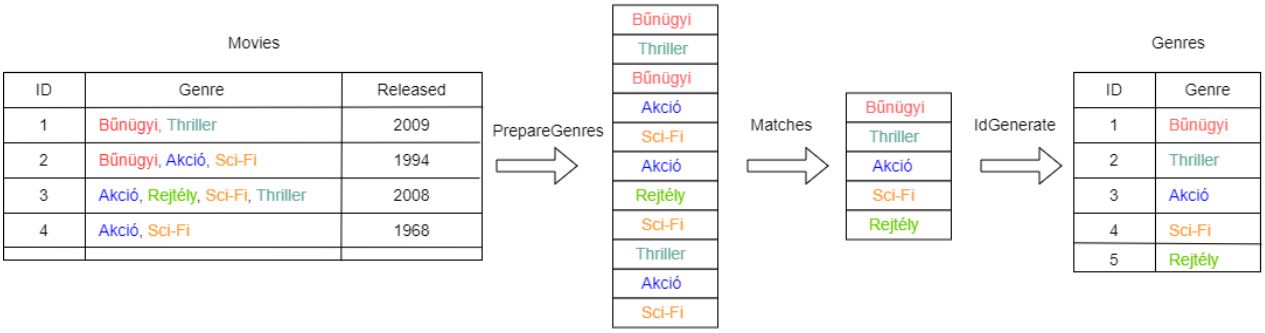
Amelyek táblát töltenek fel, azok 6 darab metódushívást tartalmaznak. Az alábbiakban a *Genre* metódus tartalma kerül bemutatásra, de a *Keyword, Language, Director, Country* metódusok is ugyanígy néznek ki, természetesen ahol szükséges ott az adott táblához igazítva:

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.



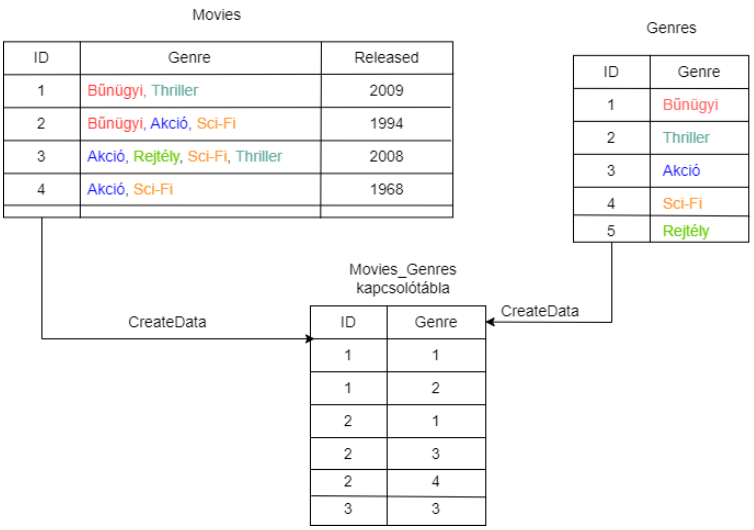
1. ábra: A Genre tábla feltöltének előkészítése

1.3.2 Kapcsolótáblát tölt fel

Azok a metódusok, amelyek kapcsolótáblát töltenek fel, azok tartalmazzák azokat a metódushívásokat is, amelyek az előbbiekben, a sima tábláknál kerültek bemutatásra. Az alábbiakban a *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 táblákból kapják, azt a 2. ábra szemlélteti.



2. 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 töltse 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.

2. 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.

2.1 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.

2.2 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, törlünk, vagy módosítunk az adatbázisunkban, vagy teljes adatbázist cserélünk tallózás segítségével, mivel ezekben az esetekben a filmekhez hozzá kell igazítanunk a többi tábla tartalmát is.

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

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

2.3 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 *SqlConnectorban* 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.

2.4 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 azonosítókon kívül a tényleges adatot is, aktuálisan a műfajok nevét. Az osztály minden ilyen adattaghoz tartalmaz egy ilyen metódust, mint jelen esetben a műfajokhoz a *GetGenre*. Ez még nem a tényleges hozzárendelést végzi el, csak az ahhoz való előkészítést. Tehát a metódusban az egyes jellemzők 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 integerek felhasználásával hozzárendeljük a
műfajok nevét is a filmekhez
        /// </summary>
        /// <param name="movies">A movies lista megfelelő adattagjában fogjuk
eltárolni a műfajok tényleges neveit is</param>
        /// <param name="tableID">A műfajok integer azonosítót tároljuk
benne</param>
        /// <param name="tableData">A műfajok neveit tároljuk benne</param>
        public void GetGenre(ref List<Movie> movies, List<int> tableID,
List<string> tableData, ref PropertiesForDecTree prop)
        {
            SQLiteDataReader reader = null;
            SQLiteCommand command = connection.CreateCommand();

            command.CommandText = "SELECT * FROM Genres";
            reader = command.ExecuteReader();
            while (reader.Read())
            {
                tableID.Add(Convert.ToInt32(reader["ID"]));
                tableData.Add(reader["Genre_Name"].ToString());
                prop.GenreAll.Add(reader["Genre_Name"].ToString());
            }
            reader.Close();
            Algorithms alg = new Algorithms();
            alg.GenreToString(ref movies,tableID,tableData);
        }
```

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 váltózó formájában. Először létrehozunk egy string típusú *structure* változót üres értékkel, majd a paraméterként kapott fájllal 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.

2.5 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. Inicilizálásuk a listák esetében az *InitListsForFillContains* metódussal, a tömbök esetében az *InitArraysForFillContains* metódussal történik.

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

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

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

```
/// <summary>
        /// A tömbök feltöltése minden egyes filmnél, azon belül minden egyes
előforduló adat esetén 0 vagy 1 értékkel
        /// </summary>
        /// <param name="prop">A PropertiesForDecTree osztály példányosításának
adattagjait töltjük fel</param>
        /// <param name="movies">A filmeken megyünk végig, illetve azok adatait
vizsgálva döntünk a 0 vagy 1 értékről</param>
        public void FillContains(ref PropertiesForDecTree prop, List<Movie>
movies)
```

```
        {

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

                        if (movies[i].GenreString[k].Contains(prop.GenreAll[j]))
                        {
                            prop.GenreContains[i][j] = 1;
                            break;
                        }
                        else
                        {
                            prop.GenreContains[i][j] = 0;
                        }
                    }
                }
            }
```

.
.
.

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

A filmeken 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 TableInsertsnek a függvénykönyvtárba szervezett programrészei is lefussanak. Tehát tartalma lényegében szinte megegyezik a TableInserts Program osztályáéval a korábban részletezett módosításoktól eltekintve.

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

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

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

2.6 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 fájlnak az elkészítése történik a fa feldolgozása közben.

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

Kezdetben az egész fára meghívjuk a metódust, szülőnek pedig a „root”-ot adjuk. Ha a *node* változó típusa *Object* (első futáskor az lesz), akkor a fának az aktuálisan vizsgált csomóponttól történő közvetlen elágazásai, illetve azoknak a további alsó (közvetett) elágazásai bekerülnek a *JArray* típusú *conditionArray* tömbbe. Tehát a metódus első futásakor az aktuálisan vizsgált csomópont az a legelső, azaz a Tmdb-score lesz, így a

conditionArray elemei a fának ettől a csomóponttól lefelé lévő részei lesznek. Mivel a Tmdb-score 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 <=6,0) esetén. Továbbá az újonnan hozzáadott csomóponthoz címkeként hozzáadjuk a csomópont nevét, jelen esetben a „Budget”-ot. Ezután újra meghívjuk a *ParseNode* metódust, mivel a Budget csomópont elágazásait hasonlóan fel kell bontani, mint azt a Tmdb-score-ral tettük. A metódus paramétereit nézve a *dot* lista ebben az esetben sem változik, a fa lebontandó része, a *JToken* az előző csomópont lebontásakor kapott kimenet lesz (tehát most a *Budget* csomópont és annak elágazásai), a *parentName* pedig az új csomópontnak (*Budget*) generált azonosító.

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

Ezzel a logikával a *ParseNode* metódussal végigmegyünk az egész json fájl tartalmán, majd a *MainJson* metódus végén a megjelenítéshez szükséges információkkal feltöltött *dot* listából készítünk egy dot fájlt, amiből GraphViz segítségével előállítjuk a megjeleníteni kívánt png képet.
Az osztály tartalmaz még egy *CreateJsonPath* metódust, ez a forrásfájl helyét határozza meg, amely szükséges a beolvasáshoz.

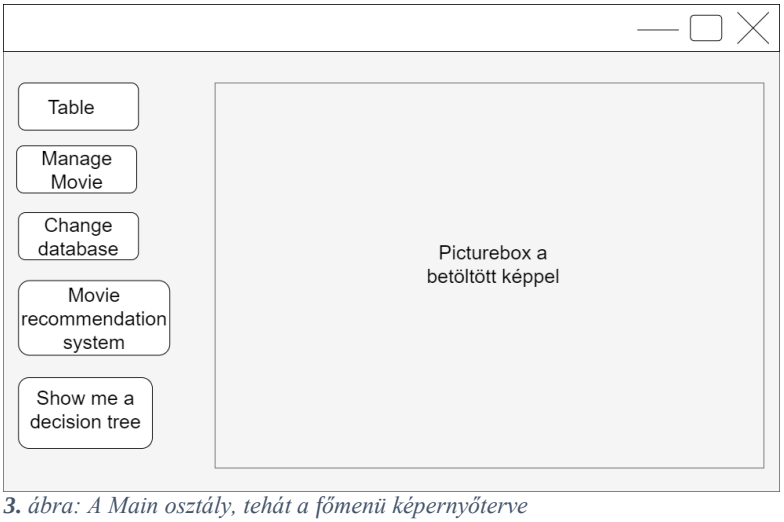
ProgressBar osztály

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

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

2.7 Main osztály

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



***3.** 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 abszolut 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 törlünk belőle, filmet módosítunk benne, vagy a teljes adatbázist cseréljük, amiből a program dolgozik.

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

2.7.1 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.

2.7.2 Change Database gomb

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

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* fájlnévvel, tehát az eddigi adatbázis fájlt fogjuk vele kicserélni. A betallózás és csere után az adatbázis tábláinak, illetve a filmek listájának újratöltését a már korábban bemutatott *Loader* metódus végzi. A metódus asszinkron

módban fut, így ez alatt az idő alatt a program megjelenít a felhasználó számára egy progress bar-t, hogy értesüljön arról, hogy az adatbázis frissítése történik.

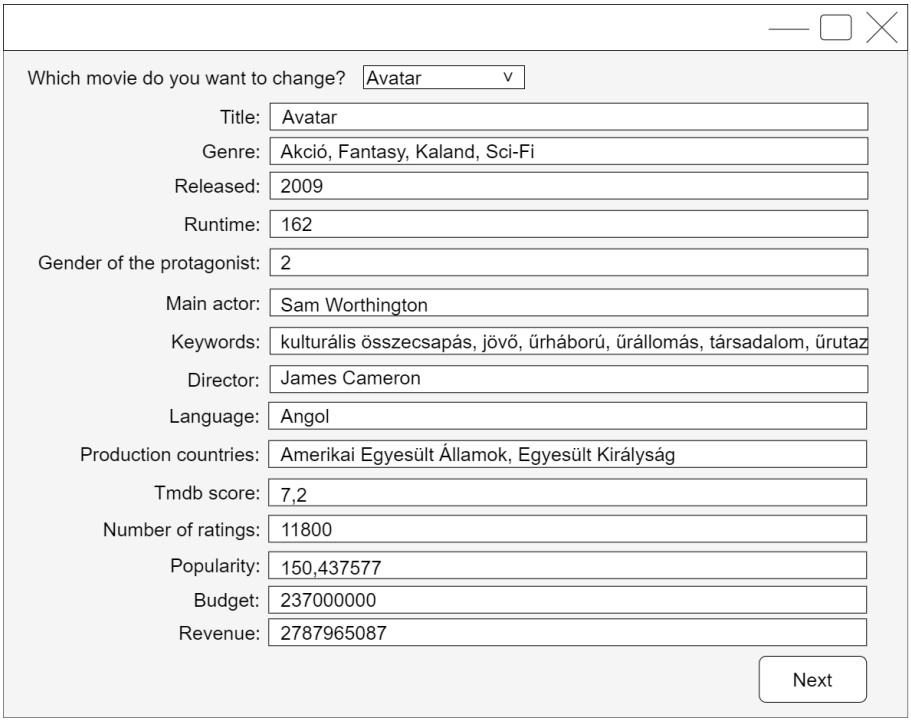
2.7.3 Manage Movie gomb

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 legördűlő lista, a Delete és a Modify gombok is, a textboxokból pedig töröljük a benne lévő szövegeket. Az eltűnt gombok helyén 2 új gombot jelenítünk meg: a Back gombot, amivel a filmek módosítását és törlését megvalósító képernyőre tudunk visszamenni, illetve az Add gombot, amit megnyomva miután a felhasználó a textboxokat feltöltötte az általa hozzáadni kívánt film adataival, a program hozzáadja azt a filmeket tartalmazó listához, és az adatbázist is hozzá igazítja. Az Add gomb megnyomása után visszatér a program az előző képernyőre, ahol a filmek adatait tudjuk módosítani, törölni.

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



4. *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.InvariantCultur
e), selectedMovie.NumberOfRatings,
selectedMovie.Popularity.ToString(System.Globalization.CultureInfo.InvariantCultu
re), 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 elrejti a szükséges grafikus elemeket, gombokat.

```
/// <summary>
        /// A módosító felület átalakítása új film beszúrásához
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void buttonAddNew_Click(object sender, EventArgs e)
        {
            cboxTitle.Visible = false;
            buttonAdd.Visible = true;
            buttonAddNew.Visible = false;
            buttonDelete.Visible = false;
            buttonModify.Visible = false;
            buttonBack.Visible = true;
            labelQuestion.Visible = false;
            labelAddNew.Visible = true;

            textBoxTitle.Text = String.Empty;
             .
             .
             .
            textBoxRevenue.Text = String.Empty;
        }
```

Az algoritmusok működéséhez segítségül szolgálnak, hasonlóképpen mint a film módosításánál a *GetFromTextboxes* és az *IsEmpty* metódusok. A *GetFromTextboxes* segítségével kiolvassuk a textboxok tartalmait és az aktuális adattag típusára alakítjuk, amelyben tárolni fogjuk, az *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 igényló mezőknél a program egy label segítségével nyújt útmutatást a felhasználó számára. A label megjelenik, ha az egér a textboxon áll, eltűnik, ha elhagyja a textboxot.

```
/// <summary>
        /// Egy címke megjelenítése ha az egér a textBoxGenre-en áll
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void textBoxGenre_MouseEnter(object sender, EventArgs e)
        {
            labelGenreHint.Visible = true;
        }
```

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

```
/// <summary>
        /// A ManageMovie osztály ablakának visszaállítása megnyitáskori
állapotára
        /// </summary>
        public void HomeScreen()
        {
            SetCboxDetails();
            FillupTextboxes();
            cboxTitle.Visible = true;
            buttonModify.Visible = true;
            buttonBack.Visible = false;
            buttonDelete.Visible = true;
            buttonAddNew.Visible = true;
            buttonAdd.Visible = false;
            labelQuestion.Visible = true;
            labelAddNew.Visible = false;
        }
```

2.8 Becslésekre vonatkozó osztályok

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

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

filmeknek, a másikban pedig az az adattag, amelynek megbecslését el szeretnénk végezni.
A fent felsorolt becslések programkódjai nem sokban különböznek egymástól, ezáltal én most csak közülük egynek a működését, felépítését fogom ismertetni, a TMDB pontszám megbecslésére irányuló *DecTreeForTmdb* osztályt, illetve a hozzá kapcsolódó 2 osztályt.

2.8.1 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.

2.8.2 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.

2.8.3 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 ismertetve lett, hogy az adatokon átalakítást kell végezni, így ezek természetesen ennél a becslésnél is szükségesek. A felhasznált tanító adatoknál szükséges az átalakítás, hiszen a *prop.GenreContains* és a *prop.KeywordContains* adattagok tartalma, amikből kiolvassuk a filmekhez tartozó megfelelő adatokat 0 vagy 1 lehet csak, ezáltal int típusú tömbök. Ezeket a kiolvasott int típusú adatokat a metódusban átalakítjuk Single-re, ami a C#-ban gyakorlatilag float típust jelent.

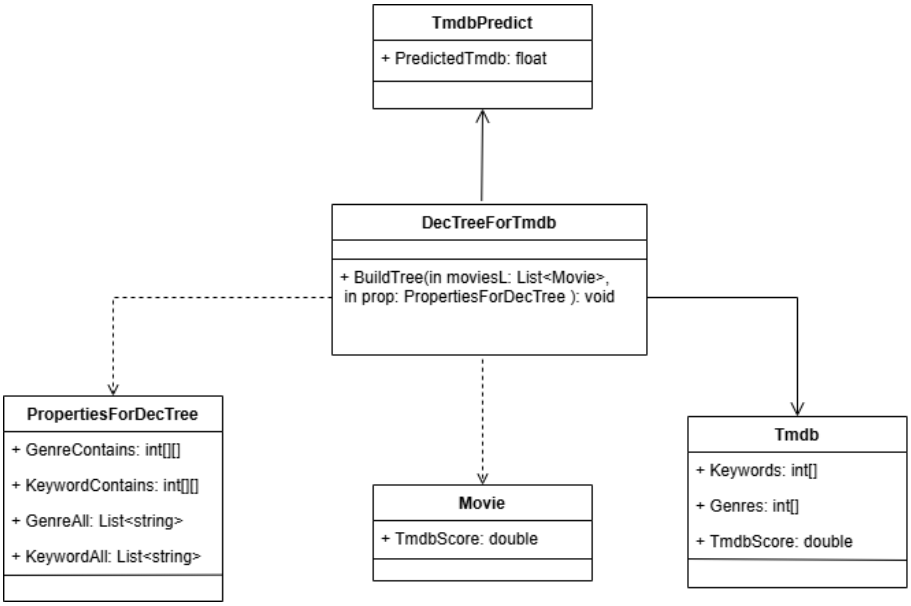
A becsléshez, mivel a TMDB pontszám egy folytonos érték, így egy regressziót használó Fasttree került felhasználásra, annak paramétereinek beállításával. Az osztály egyetlen metódust tartalmaz, az alábbi *BuildTree* metódust:

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

```
            var mlContext = new MLContext();
            var movies = new List<Tmdb>();

            for (int i = 20; i < moviesL.Count; i++)
            {
                Tmdb newTmdb = new Tmdb
                {
                    Genre = prop.GenreContains[i],
                    Keyword = prop.KeywordContains[i],
                    TmdbScore = (float)moviesL[i].TmdbScore
                };
                movies.Add(newTmdb);
            }

            var schemaDef = SchemaDefinition.Create(typeof(Tmdb));
            schemaDef["Keyword"].ColumnType = new
VectorDataViewType(NumberDataViewType.Int32, prop.KeywordAll.Count);
            schemaDef["Genre"].ColumnType = new
VectorDataViewType(NumberDataViewType.Int32, prop.GenreAll.Count);

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

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

.Append(mlContext.Transforms.Conversion.ConvertType("KeywordEncoded",
nameof(Tmdb.Keyword), DataKind.Single))
                .Append(mlContext.Transforms.Conversion.ConvertType("GenreFloat",
nameof(TMDB.Genre), DataKind.Single))
                .Append(mlContext.Transforms.Concatenate("Features",
"KeywordEncoded", "GenreFloat"))
                .AppendCacheCheckpoint(mlContext);

            var trainer = mlContext.Regression.Trainers.FastTree(
                labelColumnName: nameof(Tmdb. TmdbScore),
                featureColumnName: "Features",
                numberOfLeaves: 20,
                numberOfTrees: 150,
                minimumExampleCountPerLeaf: 8
            );

            var trainingPipeline = dataProcessPipeline.Append(trainer);
            var model = trainingPipeline.Fit(data);
            var predictions = model.Transform(data);
            var metrics = mlContext.Regression.Evaluate(predictions,
labelColumnName: nameof(Tmdb. TmdbScore));

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

            var predictionEngine = mlContext.Model.CreatePredictionEngine<Tmdb,
TmdbPredict>(model, inputSchemaDefinition: schemaDef);
            int film = 19;
            Console.WriteLine(moviesL[film].Title);
            var testMovie = new Tmdb
            {
                Genre = prop.GenreContains[film],
                Keyword = prop.KeywordContains[film]
            };
            var prediction = predictionEngine.Predict(testMovie);
```

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

A TMDB pontszámra vonatkozó becslés osztálydiagramja [5. ábra] könnyebben áttekinthetővé teszi, hogy milyen osztályok, és azoknak mely adattagjai, metódusai vesznek részt a becslés elvégzésében.



5. ábra: Osztálydiagram a TMDB pontszám becslésére vonatkozóan

2.9 MeasureAccuracy osztály

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

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

A finomhangolt verzióban is megtalálhatóak ezek a változtatások, viszont itt különböző paramétereket használunk a trainerhez finomhangolás céljából. Ezek a paraméterek például a Fasttree esetében a következőek: *numberOfLeaves*, tehát a fa leveleinek száma, a *numberOfTrees*, tehát a fák száma, a

minimumExampleCountPerLeaf, tehát a minimális példák levelenként, és a *learningRate*, tehát a tanulási ráta.

```
var trainer = mlContext.Regression.Trainers.FastTree(
                    labelColumnName: nameof(Tmdb.TmdbScore),
                    featureColumnName: "Features",
                    numberOfLeaves: leaves,
                    numberOfTrees: trees,
                    minimumExampleCountPerLeaf: minExample,
                    learningRate: learningRate
                );
```

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

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

```
.
.
.
int leaves = random.Next(2, 10);
int trees = random.Next(20, 200);
int minExample = random.Next(3, 10);
double learningRate = random.NextDouble() * (0.2 - 0.01) + 0.01;
.
.
.
if (bestRSquared < RSquare(moviesL, predictedTmdbs))
                {
                    bestRSquared = RSquare(moviesL, predictedTmdbs);
                }
.
.
.
```

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