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

Movie osztály

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

internal class Movie

{

public int Id { get; set; }

public string Title { get; set; }

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 int NumberOfRatings { get; set; }

public double Popularity { get; set; }

public long Budget { get; set; }

public long Revenue { get; set; }

/// <summary>

/// A kapcsolótáblák adatait felhasználva ezekbe az adattagokba kerülnek az egyes adatokat, pl. műfajokat azonosító integerek (id-k)

/// </summary>

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

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

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

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

public List<int> ProductionCountry { get; set; }

/// <summary>

/// Az azonosító integereket (id)-kat felhasználva ezekbe a string listákba kerülnek a filmekhez kapcsolódó tényleges adatok

/// </summary>

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;}

/// <summary>

/// Az előző listák felhasználásával a listákban szereplő adatok vesszővel történő összefűzése táblázatos megjelenítéshez

/// </summary>

public string GenreStringWithCommas

{

get => string.Join(", ", GenreString);

set

{

GenreString = value.Split(',').Select(s => s.Trim()).ToList(); // GenreString adattag frissítése a GenreStringWithCommas széttagolásával

}

}

public string KeywordStringWithCommas

{

get => string.Join(", ", KeywordString);

set

{

KeywordString = value.Split(',').Select(s => s.Trim()).ToList();

}

}

public string LanguageStringWithCommas

{

get => string.Join(", ", LanguageString);

set

{

LanguageString = value.Split(',').Select(s => s.Trim()).ToList();

}

}

public string DirectorStringWithCommas

{

get => string.Join(", ", DirectorString);

set

{

DirectorString = value.Split(',').Select(s => s.Trim()).ToList();

}

}

public string CountryStringWithCommas

{

get => string.Join(", ", CountryString);

set

{

CountryString = value.Split(',').Select(s => s.Trim()).ToList();

}

}

A MovieRecommendationSystem és a TableInserts program összekapcsolása

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

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

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

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

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

PropertiesForDecTree osztály

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

internal class PropertiesForDecTree

{

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

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

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

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

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

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

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

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

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

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

}

A listákban kerül eltárolásra az egyes tulajdonságok esetén az összes lehetséges opció, tehát például a GenreAll lista az összes előforduló műfajt tartalmazza. A listák az SqlConnectorban található Get metódusban kerülnek majd feltöltésre.

A kétdimenziós tömböknél pedig minden egyes filmet megvizsgálunk az adott jellemző minden egyes lehetséges értékére, ezek az értékek lesznek majd a tömbökben eltárolva.

SqlConnector osztály

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

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

/// <summary>

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

/// </summary>

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

public void FillupGenre(ref List<Movie> movies)

{

int index;

int data;

SQLiteDataReader reader = null;

SQLiteCommand command = connection.CreateCommand();

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

reader = command.ExecuteReader();

while (reader.Read())

{

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

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

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

{

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

{

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

}

}

}

reader.Close();

}

A kapcsolótáblában lévő, jelen esetben a műfajokat azonosító integerek megfelelőek arra, hogy hozzárendeljük a filmekhez a különböző adatokat, műfajokat, viszont a megjelenítésükkel nem derül ki, hogy konkrétan melyik műfajról van szó. Tehát szükség van egy metódusra, amely minden filmhez hozzárendeli az azonosítókon kívül a tényleges adatot is, aktuálisan a műfajok nevét. Az osztály minden ilyen adattaghoz tartalmaz egy ilyen metódust, mint jelen esetben a műfajokhoz a GetGenre. A metódusban az egyes jellemzők esetén eltároljuk az összes lehetséges értéket, ez az osztályozó modell betanításához használt tömbök feltöltéséhez lesz lényeges a későbbiekben.

/// <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()); //az összes műfaj eltárolása a prop változó GenreAll adattagjában a becslésekhez használt tömbök feltöltésének előkészítéséhez

}

reader.Close();

Algorithms alg = new Algorithms();

int mode = (int)Mode.Genre; //enumot használva beállítjuk a mode változó értékét Genre-re, amit castolással integerré alakítunk

alg.tableFiller(ref movies, tableID, tableData, mode); //a tableFiller metódusnak átadjuk a műfajok azonosítóit és a műfajokat is, a mode változó értékéből pedig tudni fogja hogy a movies lista melyik adattagját kell feltölteni a kapott adatokkal

}

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

Algorithms osztály

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

A becslésekhez használt tömb és lista adattagokat is az Algorithms osztályban inicializáljuk, illetve a tömböket itt is töltjük fel. Inicilizálásuk a listák esetében az InitListsForFillContains metódussal, a tömbök esetében a 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 szükségesek a modell betanításához.

A tömb inicializálása a fent említett metódussal már megtörtént. A feltöltéshez pedig minden egyes film esetén 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;

}

}

}

}

.

.

.

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

{

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

{

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

{

if (movies[i].CountryString[k].Contains(prop.CountryAll[j]))

{

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

break;

}

else

{

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

}

}

}

}

}

A korábbiakban említett tableFiller metódus is ebbe az osztályba került, ez dönti el, hogy a movies lista melyik adattagját töltsük fel. Az algoritmus használja a szintén az osztályban lévő enum típusú Mode-ot, melynek a swich case eseteinek azonosításában van szerepe.

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

{

switch(mode)

{

case (int)Mode.Genre:

GenreToString(ref movies, tableID, tableData);

break;

case (int)Mode.Keyword:

KeywordToString(ref movies, tableID, tableData);

break;

case (int)Mode.Language:

LanguageToString(ref movies, tableID, tableData);

break;

case (int)Mode.Director:

DirectorToString(ref movies, tableID, tableData);

break;

case (int)Mode.Country:

CountryToString(ref movies, tableID, tableData);

break;

}

}

A tableFiller minden adattag esetében másik metódust hív meg, a műfajok feltöltése esetében a GenreToString metódust.

/// <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();

}

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

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.

HandmadeLanguageDecTree osztály

A program tartalmaz egy programrészt egy kézzel készített, a film nyelvére vonatkozó döntési fa megjelenítésére is. Ennek megvalósítására ebben az osztályban kerül sor.

A döntési fa kirajzolásának alapja a ParseNode metódus, amely csomópontonként dolgozza fel a fát tartalmazó json forrásfájlt.

/// <summary>

/// Rekurzív metódus, amellyel a JSON fájlból a döntési fa csomópontonként kerül feldolgozásra

/// </summary>

/// <param name="dot">A string lista, amiből majd a dot fájl készül</param>

/// <param name="node">Az éppen aktuálisan feldolgozott csomópont</param>

/// <param name="parentName">Az adott csomópont szülője</param>

static void ParseNode(List<string> dot, JToken node, string parentName)

{

if (node.Type == JTokenType.Object)

{

var conditionArray = node["condition"] as JArray;

if (conditionArray != null)

{

foreach (var conditionObject in conditionArray)

{

var condition = conditionObject["condition"]?.ToString(); // Lekéri a feltétel szövegét

var outcome = conditionObject["outcome"]; // Lekéri a kimeneti objektumot

if (condition != null && outcome != null)

{

if (outcome.Type == JTokenType.Object) // Ha a kimenetel objektum, akkor egy új csomópont

{

string nodeName = Guid.NewGuid().ToString();

dot.Add($" \"{parentName}\" -> \"{nodeName}\" [label=\"{condition}\"];");

dot.Add($" \"{nodeName}\" [label=\"{outcome["node"]}\"];");

ParseNode(dot, outcome, nodeName);

}

else if (outcome.Type == JTokenType.Array) // Ha a kimenet egy tömb, akkor minden elemet külön kezel

{

foreach (var item in outcome)

{

string leafName = Guid.NewGuid().ToString();

dot.Add($" \"{parentName}\" -> \"{leafName}\" [label=\"{condition}\"];");

dot.Add($" \"{leafName}\" [label=\"{item}\", shape=box];");

}

}

else

{

string leafName = Guid.NewGuid().ToString();

dot.Add($" \"{parentName}\" -> \"{leafName}\" [label=\"{condition}\"];");

dot.Add($" \"{leafName}\" [label=\"{outcome}\", shape=box];");

}

}

}

}

}

else if (node.Type == JTokenType.Array) //Ha az aktuális csomópont tömb, akkor szintén minden elemet külön dolgoz fel

{

foreach (var item in node)

{

ParseNode(dot, item, parentName);

}

}

else

{

string leafName = Guid.NewGuid().ToString();

dot.Add($" \"{parentName}\" -> \"{leafName}\" [label=\"\"];");

dot.Add($" \"{leafName}\" [label=\"{node}\", shape=box];");

}

}

A ParseNode metódus a MainJson metódusból kerül meghívásra. Ebben a metódusban történnek az alapvető változó deklarációk, a json forrásfájl beolvasása, json objektummá való átalakítása, illetve a dot string típusú lista létrehozása is, amiből majd kezdetben a .dot fájl készül, később pedig ebből a megjelenítendő .png kép.

/// <summary>

/// A döntési fa felépítéséhez szükséges változók létrehozása, függvények meghívása

/// </summary>

public void MainJson()

{

string json = CreateJsonPath();

json = File.ReadAllText(json);

JObject jsonObj = JObject.Parse(json);

var dot = new List<string>

{

"digraph DecisionTree {"

};

dot.Add($" \"root\" [label=\"{jsonObj["decision\_tree"]["root"]["node"]}\"];");

ParseNode(dot, jsonObj["decision\_tree"]["root"], "root");

dot.Add("}");

File.WriteAllLines("decision\_tree.dot", dot); //A dot string lista tartalmából készít egy .dot fájlt

ProcessStartInfo startInfo = new ProcessStartInfo //Graphviz parancsok amelyek segítségével a .dot fájlból .png kép készül

{

FileName = "dot",

Arguments = "-Tpng decision\_tree.dot -o decision\_tree.png",

RedirectStandardOutput = true,

UseShellExecute = false,

CreateNoWindow = true

};

using (Process process = Process.Start(startInfo)) //A folyamat indítása

{

process.WaitForExit();

}

PictureBox pictureBox = new PictureBox //PictureBox használata a kép megjelenítéséhez

{

Dock = DockStyle.Fill,

Image = Image.FromFile("decision\_tree.png"),

SizeMode = PictureBoxSizeMode.Zoom

};

this.Controls.Add(pictureBox);

}

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

Main osztály

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

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

A Main osztály egy további metódusa, a CreatePath építi fel az adatbázis eléréséhez szükséges utat, ezzel elkerülhető az abszolut útvonal használata. Ez a metódus is a Main\_Load-ból kerül meghívásra.

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

Az osztály 4 adattagot tartalmaz: a Movie típusú movies listát, amelyben a filmek adatait tároljuk, az int típusú tableID és string típusú tableData, továbbá a MovieTI típusú moviesTI listát (a TI rövidítés a TableInserts program osztályait jelöli a névütközés elkerülése érdekében), amelyek a TableInsertsből származó programrészek lefutásához szükségesek.

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.

/// <summary>

/// A táblák, listák értékeinek újragenerálását és feltöltését végzi abban az esetben, ha módosul az adatbázis

/// </summary>

public void Loader()

{

movies.Clear();

string databasePath = CreatePath();

SqlConnector conn = new SqlConnector(databasePath);

Algorithms alg = new Algorithms();

HandmadeLanguageDecTree tree = new HandmadeLanguageDecTree();

PropertiesForDecTree properties = new PropertiesForDecTree();

alg.LoadTableInserts(ref moviesTI, databasePath); // A TableInserts program metódushívásait összefogó metódus meghívása

movies = conn.collectMovie(); // a filmeket tároló lista újra feltöltése már naprakész adatokkal

alg.InitLists(ref movies);

alg.InitListsForFillContains(ref properties);

conn.FillupGenre(ref movies); // A korábbiakban már részletezett metódusok meghívása

conn.FillupKeywords(ref movies);

conn.FillupDirector(ref movies);

conn.FillupLanguage(ref movies);

conn.FillupCountry(ref movies);

conn.GetGenre(ref movies, tableID, tableData, ref properties);

conn.GetKeyword(ref movies, tableID, tableData, ref properties);

conn.GetDirector(ref movies, tableID, tableData, ref properties);

conn.GetLanguage(ref movies, tableID, tableData, ref properties);

conn.GetCountry(ref movies, tableID, tableData, ref properties);

alg.InitArraysForFillContains(ref properties, movies);

alg.FillContains(ref properties, movies);

Table gomb

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

/// <summary>

/// A Table gomb megnyomásakor lefutó metódus

/// </summary>

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

{

ShowTable showTable = new ShowTable(movies);

showTable.Show();

}

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

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

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

Change Database gomb

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

/// <summary>

/// A Change Database gomb megnyomásának hatására lefutó metódus, ez végzi el az adatbázis cseréjét

/// </summary>

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

{

OpenFileDialog openFileDialog = new OpenFileDialog(); // Fájltallózó intéző ablak létrehozása

openFileDialog.Title = "Select the database file";

DialogResult result = openFileDialog.ShowDialog(); // A létrehozott ablak megjelenítése

if (result == DialogResult.OK) // Ha a felhasználó kiválasztja az adott betallózandó fájlt

{

string source=openFileDialog.FileName; // Eltárolja a választott fájl elérési útvonalát

string destination = AppDomain.CurrentDomain.BaseDirectory;

string destinationFilePath = Path.Combine(destination, "movies.db"); //Beállítja célútvonalnak a program saját könyvtárát, és a bemásolandó fájl nevét movies.db-re

try

{

File.Copy(source, destinationFilePath, true); // A betallózott fájlt eredeti helyéről a program saját könyvtárába másolja, ezzel lecserélve a régi movies.db fájlt

}

catch

{

MessageBox.Show("Error");

}

ProgressBar progressBar = new ProgressBar();

progressBar.Show(); // Progress bar megjelenítése ami jelzi a felhasználó felé, hogy az adatbázis frissítése történik

try

{

await Task.Run(() => Loader()); // Aszinkron módban futtatva meghívja az osztály Loader() metódusát annak érdekében, hogy a feltöltés alatt ne fagyjon meg a program

}

finally

{

progressBar.Visible = false; //Ha lefutott a Loader metódus, tehát lényegében a lista és adatbázis frissítés, akkor a progress bar eltűnik

}

}

}

ProgressBar osztály

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

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

Manage Movie gomb

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

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

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

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

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

Az ablak megnyílásának eseményére a ManageMovie osztály ManageMovie\_Load metódusa fut le. Ebben egy egyszerű metódushívás történik, a SetCboxDetails metódust hívja meg. Ez a metódus az ablak működésének alapját adó combobox tulajdonságait állítja be.

/// <summary>

/// Combobox tulajdonságainak beállítása

/// </summary>

private void SetCboxDetails()

{

cboxTitle.DataSource = moviesForModify; // A combobox adatforrása a moviesForModify lista

cboxTitle.DisplayMember = "Title"; // A comboboxot lenyitva a filmek címei jelennek meg

cboxTitle.ValueMember = "Id"; // A filmeket a comboboxban is az Id-juk segítségével azonosítjuk

}

Ha a combobox, azaz a legördülő listában megváltozik a választott film, akkor a cboxTitle\_SelectedIndexChanged metódus fut le. Ennek a metódusnak a tartalma szintén egy egyszerű metódushívás, a FillupTextboxes metódust hívja meg. Ez a textboxokat feltölti a legördülő listából kiválasztott film megfelelő adataival.

/// <summary>

/// A textboxok feltöltése a kiválasztott film adataival

/// </summary>

private void FillupTextboxes()

{

selectedMovie = (Movie)cboxTitle.SelectedItem; // A selectedMovie változóban eltároljuk a kiválasztott film adatait referencia szerint

if (selectedMovie != null)

{

textBoxTitle.Text = selectedMovie.Title;

textBoxGenre.Text = selectedMovie.GenreStringWithCommas;

textBoxReleased.Text = selectedMovie.Released.ToString();

textBoxRuntime.Text = selectedMovie.Runtime.ToString();

textBoxGenderOfTheProtagonist.Text = selectedMovie.GenderOfProtagonist.ToString();

textBoxMainActor.Text = selectedMovie.MainActor;

textBoxKeywords.Text = selectedMovie.KeywordStringWithCommas.ToString();

textBoxDirector.Text = selectedMovie.DirectorStringWithCommas.ToString();

textBoxLanguage.Text = selectedMovie.LanguageStringWithCommas.ToString();

textBoxCountries.Text = selectedMovie.CountryStringWithCommas.ToString();

textBoxTmdbScore.Text = selectedMovie.TmdbScore.ToString();

textBoxNumberOfRatings.Text = selectedMovie.NumberOfRatings.ToString();

textBoxPopularity.Text = selectedMovie.Popularity.ToString();

textBoxBudget.Text = selectedMovie.Budget.ToString();

textBoxRevenue.Text = selectedMovie.Revenue.ToString();

}

}

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. Amennyiben nincs inputhiba, akkor az adatbázisban is frissíti az adott film adatait a selectedMovie változó adatait felhasználva. A frissítéshez az SqlConnector UpdateCommand metódusát használja fel. 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.

/// <summary>

/// A Modify gomb hatására végrehajtandó algoritmus

/// </summary>

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

{

if (IsEmpty() == false) // Mezők ürességének ellenőrzése

{

GetFromTextboxes(ref selectedMovie); // A mezőkből az adatok kiolvasása

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); // Az összekészített SQL utasítás végrehajtása

Main main = new Main();

ProgressBar progressBar = new ProgressBar();

progressBar.Show();

try

{

await Task.Run(() => main.Loader()); // Futtatás aszinkron módban a fagyás kiküszöbölésének érdekében

}

finally

{

progressBar.Visible = false;

}

}

}

}

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

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

/// <summary>

/// A módosító felület átalakítása új film beszúrásához

/// </summary>

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

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

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

{

cboxTitle.Visible = false;

buttonAdd.Visible = true;

buttonAddNew.Visible = false;

buttonDelete.Visible = false;

buttonModify.Visible = false;

buttonBack.Visible = true;

labelQuestion.Visible = false;

labelAddNew.Visible = true;

textBoxTitle.Text = String.Empty; // A textboxok tartalmának kiürítése

textBoxGenre.Text = String.Empty;

textBoxReleased.Text = String.Empty;

textBoxRuntime.Text = String.Empty;

textBoxGenderOfTheProtagonist.Text = String.Empty;

textBoxMainActor.Text = String.Empty;

textBoxKeywords.Text = String.Empty;

textBoxDirector.Text = String.Empty;

textBoxLanguage.Text = String.Empty;

textBoxCountries.Text = String.Empty;

textBoxTmdbScore.Text = String.Empty;

textBoxNumberOfRatings.Text = String.Empty;

textBoxPopularity.Text = String.Empty;

textBoxBudget.Text = String.Empty;

textBoxRevenue.Text = String.Empty;

}

Az algoritmusok működéséhez segítségül szolgálnak a GetFromTextboxes és az IsEmpty metódusok.

/// <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.Runtime = int.Parse(textBoxRuntime.Text);

selectedMovie.GenderOfProtagonist = int.Parse(textBoxGenderOfTheProtagonist.Text);

selectedMovie.MainActor = textBoxMainActor.Text;

selectedMovie.KeywordStringWithCommas = textBoxKeywords.Text;

selectedMovie.DirectorStringWithCommas = textBoxDirector.Text;

selectedMovie.LanguageStringWithCommas = textBoxLanguage.Text;

selectedMovie.CountryStringWithCommas = textBoxCountries.Text;

selectedMovie.TmdbScore = double.Parse(textBoxTmdbScore.Text);

selectedMovie.NumberOfRatings = int.Parse(textBoxNumberOfRatings.Text);

selectedMovie.Popularity = double.Parse(textBoxPopularity.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;

}

}

}

/// <summary>

/// A textboxok tartalmának ellenőrzése abból a szempontból, hogy üres-e bármelyik

/// </summary>

/// <returns>Az ellenőrzés eredményét adja vissza: igaz esetén van üres, hamis esetén nincs</returns>

public bool IsEmpty()

{

bool status=false;

if (string.IsNullOrEmpty(textBoxTitle.Text)|| string.IsNullOrEmpty(textBoxGenre.Text)|| string.IsNullOrEmpty(textBoxReleased.Text) || string.IsNullOrEmpty(textBoxRuntime.Text)

|| string.IsNullOrEmpty(textBoxGenderOfTheProtagonist.Text ) || string.IsNullOrEmpty(textBoxMainActor.Text) || string.IsNullOrEmpty(textBoxKeywords.Text)

|| string.IsNullOrEmpty(textBoxDirector.Text) || string.IsNullOrEmpty(textBoxLanguage.Text) || string.IsNullOrEmpty(textBoxCountries.Text)

|| string.IsNullOrEmpty(textBoxTmdbScore.Text) || string.IsNullOrEmpty(textBoxNumberOfRatings.Text) || string.IsNullOrEmpty(textBoxPopularity.Text)

|| string.IsNullOrEmpty(textBoxBudget.Text) || string.IsNullOrEmpty(textBoxRevenue.Text))

{

status = true;

MessageBox.Show("There are empty fields!", "Error",

MessageBoxButtons.OK, MessageBoxIcon.Error);

}

return status;

}

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)

{

// Amikor az egér a TextBox fölé ér, megjelenítjük a label-t

labelGenreHint.Visible = true;

}

/// <summary>

/// A címke eltüntetése ha az egér már nem áll a textBoxGenre-ön

/// </summary>

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

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

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

{

// Amikor az egér elhagyja a TextBox-ot, elrejtjük a label-t

labelGenreHint.Visible = false;

}

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

/// <summary>

/// A ManageMovie osztály ablakának visszaállítása megnyitáskori állapotára

/// </summary>

public void HomeScreen()

{

SetCboxDetails();

FillupTextboxes();

cboxTitle.Visible = true;

buttonModify.Visible = true;

buttonBack.Visible = false;

buttonDelete.Visible = true;

buttonAddNew.Visible = true;

buttonAdd.Visible = false;

labelQuestion.Visible = true;

labelAddNew.Visible = false;

}

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

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

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

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

**Tmdb osztály**

internal class Tmdb

{

public int[] Keyword { get; set; }

public int[] Genre { get; set; }

public double TmdbScore { get; set; }

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

**TmdbPredict osztály**

internal class TmdbPredict

{

[ColumnName("Score")]

public float PredictedTmdb { get; set; }

}

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

**DecTreeForTmdb**

Ez az az osztály, ahol a tanító adathalmaz megadása történik, a modell felépítése, betanítása, az adott film kiválasztása, aminek a pontszámát szeretnénk megbecsülni, illetve maga a becslés is. Egyetlen metódust tartalmaz, az alábbi BuildTree metódust:

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

{

var mlContext = new MLContext();

//A tanító adatokat tartalmazó lista létrehozása és feltöltése, ez esetben az adatbázis első 20 darab film megfelelő adataival

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);

}

// Egyéni séma definíció létrehozása annak érdekében, hogy dinamikus legyen a vektor mérete

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);

// Adatok betöltése az egyéni séma használatával

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

// A betöltött adatok átalakítása megfelelő formába

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);

// A modell típusának beállítása, amely ebben az esetben regressziót használó FastTree, és paramétereinek értékének beállítása

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); // Modell tanítása a Fit() metódussal

// A betanított modell kiértékelése

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}");

// Egy darab film Tmdb pontszámának megbecslése a modell használatával

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

int film = 19;

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

var testMovie = new Tmdb // A becsléshez használt adattagok feltöltése a megbecsülendő film megfelelő adataival

{

Genre = prop.GenreContains[film],

Keyword = prop.KeywordContains[film]

};

var prediction = predictionEngine.Predict(testMovie); // A tényleges becslés elvégzése

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

}

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.

**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); // numberOfLeaves érték generálása: 2-től 30-ig int trees = random.Next(20, 200); // numberOfTrees érték generálása: 20-tól 200-ig

int minExample = random.Next(3, 10); // minimumExampleCountPerLeaf generálása: 1-től 10-ig

double learningRate = random.NextDouble() \* (0.2 - 0.01) + 0.01; //learningRate generálása: 0.01-től 0.2-ig

.

.

.

if (bestRSquared < RSquare(moviesL, predictedTmdbs))

{

bestRSquared = RSquare(moviesL, predictedTmdbs);

}

.

.

.

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

/// <summary>

/// R^2 érték számítás teljes adathalmazra a modellek hatékonyságának méréséhez

/// </summary>

/// <param name="originalMovies">A filmek eredeti adatait, köztük a Tmdb pontszámokat is tartalmazó lista</param>

/// <param name="predictedValues">A becsült Tmdb pontszámokat tartalmazó lista</param>

/// <returns></returns>

public float RSquare(List<Movie> originalMovies, List<float> predictedValues)

{

float RSq = 0;

float mean = 0;

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

{

mean += (float)originalMovies[i].TmdbScore;

}

mean = (mean / originalMovies.Count);

float counter = 0;

float denominator = 0;

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

{

counter += (float)Math.Pow(((float)originalMovies[i].TmdbScore - predictedValues[i]), 2);

denominator += (float)Math.Pow(((float)originalMovies[i].TmdbScore - mean), 2);

}

RSq = 1 - (counter / denominator);

return RSq;

}

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

Recommendation System gomb

**SetPriority osztály**

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

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

Az osztály működésének koncepciója, hogy létrehozunk 2 PriorityListItem típusú listát. A PriorityList osztály a következő képpen néz ki:

internal class PriorityListItem

{

public int Id { get; set; }

public string Title { get; set; }

public int Priority { get; set; }

public override string ToString() // A ToString() metódus felülírása annak érdekében, hogy ha később hivatkozunk egy PriorityListItem lista egy-egy elemére, akkor a Title tulajdonsága alapján beazonosítható legyen

{

return Title;

}

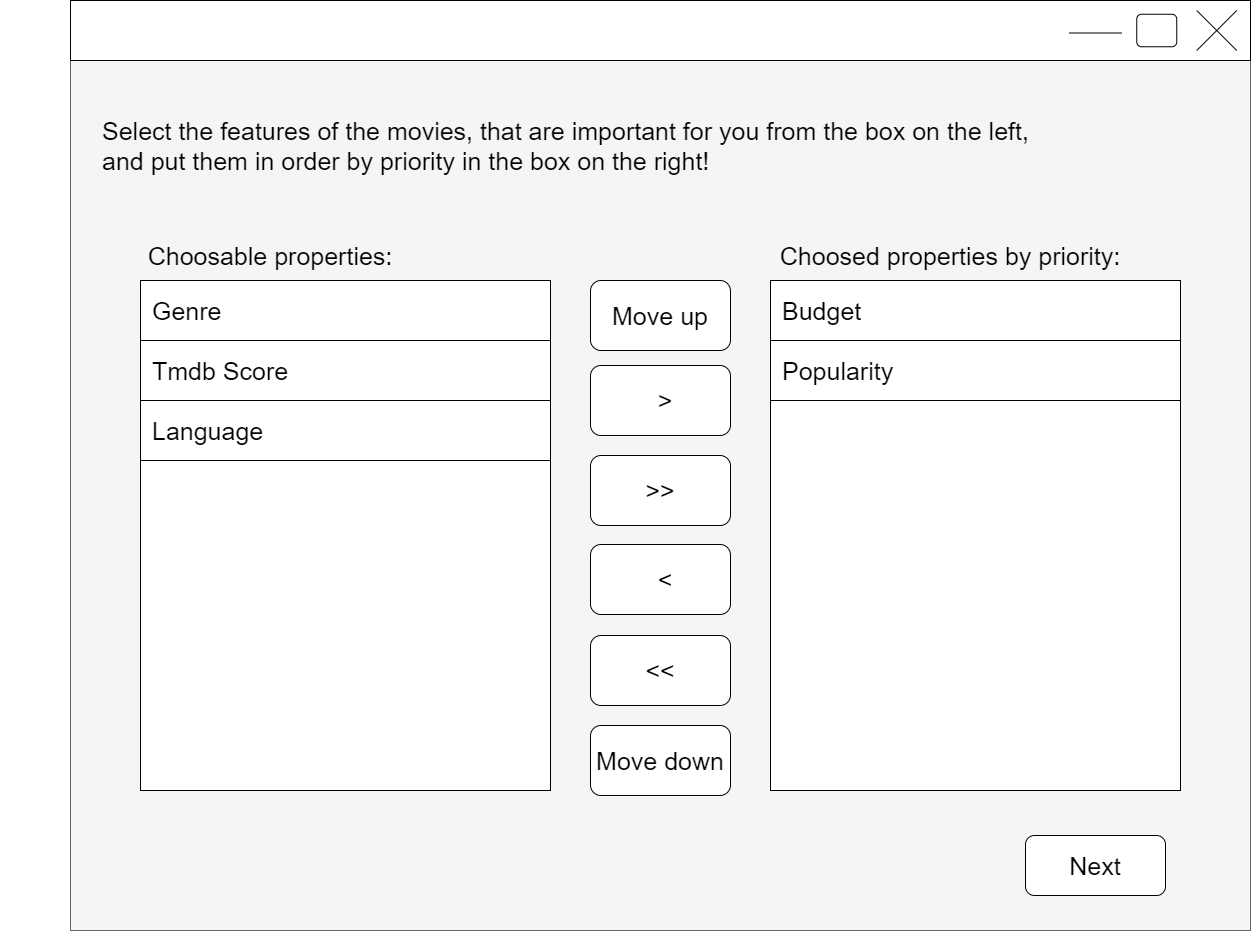
}

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

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

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

Az osztályhoz tartozó ablak a következő képpen néz ki:



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

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

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

A „>” gomb algoritmusa a következő:

private void MoveSelectedItem()

{

if (listBoxOriginal.SelectedItem != null) //Ellenőrizzük hogy van-e kiválasztva elem

{

PriorityListItem selectedItem = (PriorityListItem)listBoxOriginal.SelectedItem; //A kiválasztott elem eltárolása

originalList.Remove(selectedItem); //Az elem törlése korábbi helyéről

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

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

}

}

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

/// <summary>

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

/// </summary>

private void UpdatePriorities()

{

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

{

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

}

}

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

/// <summary>

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

/// </summary>

private void MoveUp()

{

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

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

{

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

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

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

RefreshListBoxes(); // A listboxok frissítése

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

}

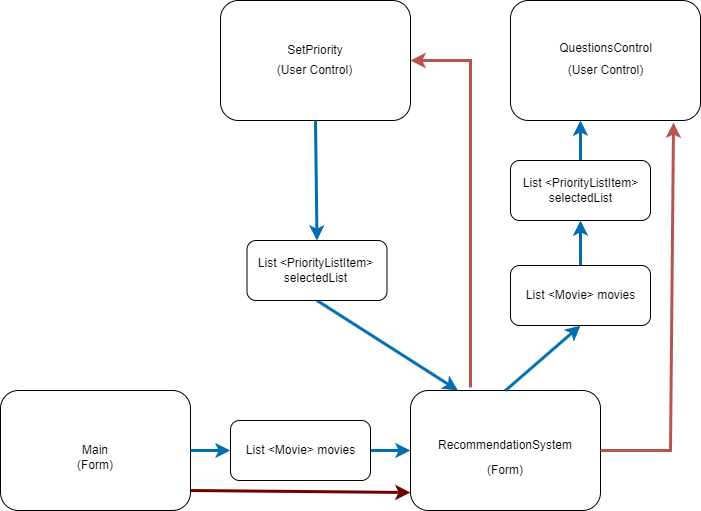
}

Lenne egy PriorityList típusú listám, amelyben a listaelemeknek 3 adattagja lenne: egy id, és egy string Title, és egy int Priority prioritási szám (mivel a felhasználó döntene a prioritásokról, ezért kezdetben mindegyiknek 0 lenne). A lista elemeit szeretném megjeleníteni egy listboxban, az elemek Title adattagja lenne az a szöveg ami megjelenítésre kerül objektumonként a listboxban. Lenne egy másik listboxom is, amihez egy üres, de ugyanúgy PriorityList típusú lista tartozna hozzá. A felhasználó által az eredeti listát tartalmazó listboxból kiválasztott elemek egyesével kerülnének át az új listboxba a Move gomb megnyomása után. Ha az össze elemet szeretné átvinni, akkor a MoveAll gombbal az összes elem átmozgatható. Végül az új listboxban a bekerült elemek felfelé, és lefelé is mozgathatóak lennének a Move Up és a Move Down gombokkal. Az új listboxhoz, amibe átmozgatta a felhasználó az elemeket, ahhoz is tartozna egy PriorityList típusú lista, amelyben az elemek olyan prioritási értéket kapnának, amilyen sorrendben elhelyezkednek a listboxban. Tehát a legfelső elem Priority adattagja 0 lenne, az alatta lévő elemé 1, és így tovább. Ha az eredeti listboxból átkerülne az elem az új listboxba, akkor az eredetiből értelemszerűen törölni kell, hogy ne legyen egyszerre 2 helyen. Illetve kellene egy Move back gomb is, amivel az új listboxból visszahelyezhetőek az elemek az eredetibe, ha a felhasználó esetleg meggondolta magát az elemek kiválasztása közben.

**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 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.

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 kérdésekben.



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

