**ML.NET**

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

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

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

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

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

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

Mi az az ML.NET?

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

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

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

Hogyan néz ki egy ML.NET alkalmazás?

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

* MLContext létrehozása

Egy ML.NET alkalmazás legelső lépése az MLContext létrehozása, hiszen ezek után kezdődhet csak el az adatok betöltése, átalakítása, a modell betanítása, az algoritmusok kiválasztása. Tehát az MLContext az egész ML.NET alkalmazás kiindulópontja.

var mlContext = new MLContext();

* Adatok betöltése

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

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

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

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

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

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

* Adatok átalakítása

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

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

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

Az alábbi kódrészlet az adatok átalakítására mutat egy példát:

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

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

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

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

.AppendCacheCheckpoint(mlContext);

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

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

* Algoritmus kiválasztása

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

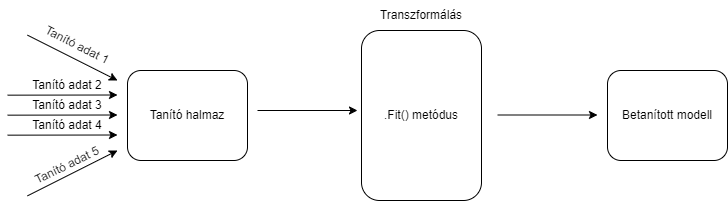
Az ML.NET által támogatott algoritmusok között megtalálhatóak a következő típusú metódusok:

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

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

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

ITransformer model = pipeline.Fit(trainingData);



* Modell kiértékelése

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

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

A többosztályos osztályozó esetében az alábbi értékek használhatóak a kiértékelésre:

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

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

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

var predictions = model.Transform(data);

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

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

Regressziót használó osztályozók esetében az alábbi módszerek használhatóak a kiértékelésre:

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

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

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

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

var model = trainingPipeline.Fit(data);

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

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

A modell kimentése a fenti változók segítségével:

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

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

DataViewSchema modelSchema;

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

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

* Algoritmusok hatékonyságának összehasonlítása

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

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

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

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

A kapott eredmények R^2 pontosságai a következőek voltak:

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

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

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

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

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

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

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

Források:

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

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

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

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

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

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

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

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

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

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