

# Járművek trajektóriájának előrejelzése machine learning modellekkel

PÉTER BENCE MÉRNÖKINFORMATIKA BSC 6. FÉLÉV\*, Széchenyi István Egyetem, Hungary  
DR. HORVÁTH ANDRÁS, Széchenyi István Egyetem, Hungary  
AGG ÁRON PHD HALLGATÓ, Széchenyi István Egyetem, Hungary



**SZÉCHENYI  
EGYETEM**  
UNIVERSITY OF GYŐR



**INFORMATIKA  
TANSZÉK**  
DEPARTMENT OF COMPUTER SCIENCE

Az ITS (intelligent transportation system) egyre nagyobb teret hódít napjainkban és rengeteg különböző területen alkalmazzák ezeket a rendszereket. A közlekedési csomópontok elemzése egy frekventált terület az ITS alkalmazásában. Célunk, gépi látás és gépi tanulás felhasználásával, közlekedési csomópontok elemzésének automatizálása és felgyorsítása. A kutatásban lefektetett alapgondolatokat, kifejlesztett keretrendszerét és a felmerülő problémák megoldásait, a gyakorlatban balesetek megelőzésére, renitens viselkedések kiszűrésére és forgalomirányító rendszerek támogatására lehet használni. A kutatásban egy trajektória

Authors' addresses: Péter Bence Mérnökinformatika BSc 6. félév, Széchenyi István Egyetem, Győr, Hungary; Dr. Horváth András, Széchenyi István Egyetem, Győr, Hungary; Agg Áron PhD hallgató, Széchenyi István Egyetem, Győr, Hungary.

2023. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in , <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>.

osztályozó módszert ismertetünk, amely objektumdetektálás és objektumkövetés segítségével elemezi a közlekedési csomópontokban elhaladó járművek mozgását. A mozgásuk alapján klaszterezí a trajektóriákat, majd gépi tanulás segítségével predikciót ad az újonnan belépő járművek kilépési pontjára. A módszerhez 6 különböző közlekedési csomópontban készített saját video adatbázisunkat használtuk fel. A tesztelt klaszterezési módszerek közül (OPTICS, BIRCH, KMeans, DBSCAN) az OPTICS algoritmus bizonyult legjobbnak trajektórák klaszterezésére. Összehasonlítottunk több különböző klasszifikációs módszert a legfontosabb predikció eléréséhez, amelyek: KNN, SVM, GP, DT, GNB, MLP, SGD. A tanulmányban bemutatott eljárások közül az KNN adta átlagban a legfontosabb 90%-os eredményt.

## ACM Reference Format:

Péter Bence Mérnökinformatika BSc 6. félév, Dr. Horváth András, and Agg Áron PhD hallgató. 2023. Járművek trajektóriájának előrejelzése machine learning modellekkel. 1, 1 (April 2023), 15 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## CONTENTS

|                                |    |
|--------------------------------|----|
| Abstract                       | 1  |
| Contents                       | 2  |
| 1 Bevezetés                    | 2  |
| 2 Kapcsolódó kutatások         | 3  |
| 2.1 YOLO                       | 4  |
| 2.2 DeepSORT                   | 5  |
| 3 Adathalmazok kialakítása     | 5  |
| 3.1 Adatstruktúra              | 6  |
| 3.2 Objektumdetektálás         | 6  |
| 3.3 Objektumkövetés            | 6  |
| 4 Klaszterezés                 | 6  |
| 4.1 Adattisztítás              | 7  |
| 4.2 Feature vektorok           | 7  |
| 4.3 Klaszterezési algoritmusok | 7  |
| 4.4 Paraméterek kiválasztása   | 10 |
| 5 Klasszifikáció               | 10 |
| 5.1 Multiclass                 | 10 |
| 5.2 Binary                     | 10 |
| 5.3 OneVsRest                  | 10 |
| 5.4 Machine Learning modellek  | 10 |
| 5.5 Feature vektorok           | 12 |
| 5.6 Pontosság mérése           | 13 |
| 6 Valós idejű alkalmazás       | 14 |
| 7 Konklúzió                    | 14 |
| References                     | 15 |

## 1 BEVEZETÉS

A városok növekedése egyre nagyobb forgalomhoz vezet, ami a balesetek, forgalmi dugók számát növeli és a levegő minősége is romlik.

Az ITS (*intelligent transportation system*). fejlesztése a városokban erre megoldást jelenthet. Ez magába foglalja az információs és kommunikációs technológiák, mint például szenzorok, kamerák, kommunikációs hálózatok és adat elemzés fejlesztését. 5G hálózatokon keresztül, ezek a technológiák összeköthetők a közlekedési eszközökkel. Ehhez okos forgalomirányítási rendszerek kifejlesztésére van szükség, amik információval tudnak szolgálni a járművekbe szerelt informatikai rendszereknek. A legértékesebb információt a közlekedésben részvevő járművek jelen és jövőbeli pozíciója jelenti. Pontos és gyors trajektória előrejelző rendszerek kifejlesztése egy nagy kihívás és egyre növekszik irántuk a kereslet. E kutatási terület kiforrasztásából eredően, kevés létező keretrendszer és adathalmaz található, így a tanító adathalmaz gyűjtése, adatok kinyerésének formátuma, tárolása és mérőszámok kifejlesztése (amivel a tesztelni kívánt modellek pontosságát tudjuk mérni) is a kutatáshoz tartoznak. Ebben a kutatásban erre a problémára törekszünk egy módszertant és keretrendszer kifejleszteni, emellett klaszterezési és klasszifikációs gépi tanulási algoritmusokat tesztelni.

*Machine Learning.* A gépi tanulás számos különböző típusa létezik, például a felügyelt tanulás, a felügyelet nélküli tanulás és a megerősítő tanulás. A felügyelt tanulásban a modell az adatokon keresztül próbál megtanulni egy adott feladatot. A modellnek az adatok mellett ismert kimeneti értékekre van szüksége, amelyek segítik a modell tanulását és az előrejelzéseket. A felügyelet nélküli tanulásban a modellnek az adatokból kell megtalálnia a mintákat és összefüggéseket anélkül, hogy előzetesen ismert kimeneti értékekre támaszkodna. A megerősítő tanulásban a modell az adatokon és a rendszeren keresztül próbál megtanulni, és visszajelzést kap a teljesítményéről. A gépi tanulás nagyon széles körben alkalmazható, például az automatikus beszédfelismerésben, a képfelismerésben, a termékjánlásokban, a pénzügyi előrejelzésekben, az egészségügyben és az üzleti elemzésekben. Az adatok rendelkezésre állása miatt az iparágak és a kutatási területek számos területen használják a gépi tanulást az előrejelzések és a döntéshozatal támogatása érdekében. Mi esetünkben forgalomban résztvevő objektumok trajektoriájának osztályozáshoz használjuk ezeket a gépi tanulási algoritmusokat. A forgalomban fellelhető szabályosságokat, unsupervised tanulási módszerrel, úgynevet klaszterezéssel határozzuk meg. Erre a feladatra KMeans, BIRCH [Zhang et al. 1996] és DBSCAN [Ester et al. 1996][Schubert et al. 2017], OPTICS [Ankerst et al. 1999] algoritmusokat teszteltük. A klaszterezés során az objektumok be- és kimeneti pontjai szolgálnak bemenetként az algoritmusoknak, az algoritmusok által meghatározott trajektória klaszterek lesznek a klasszifikáció tanítására felhasznált osztályok. A klaszterezési lépés felgyorsítja a klasszifikációs modellek tanítását, mivel a trajektoriák osztályokba sorolását kézzel is el lehetne végezni, ami nagy adathalmazok esetén nagyon hosszú idő lenne. A klasszifikáció egy supervised tanulási módszer, amihez mi bináris klasszifikációs modelleket kombinálunk, ami magas osztályszámnál, ami a mi esetünkben átlagosan 10-15 között volt, igen hatékony. minden bináris modellnél, egy osztály az összes többivel szemben van betanítva. A modellek pontosságának kiértékelésére 3 mérőszámot alkalmaztunk, amik az *Accuracy Score*, *Balanced Accuracy Score* [Brodersen et al. 2010] és *Top-k Accuracy Score*. Mindegyik mérőszám kiszámolásához *K-Fold Cross-Validation* [Anguita et al. 2012] metódust alkalmaztunk, ahol  $K = 5$ .

*A tanító adatok* előállításához, objektumok detektálására a YOLOv7 [Wang et al. 2022] konvoluciós neurális hálót használtuk, ez a konvoluciós neurális háló architektúra nem csak nagy pontosságot hanem sebességet is nyújt nekünk. Emellett képkockáról képkockára követni is kell tudni a detektált objektumokat. Erre is sok megoldás található manapság, erre a feladatra a DeepSORT [Wojke and Bewley 2018] nevezetű algoritmust használtuk, ez kálmán filtert és konvoluciós neurális hálót használ az objektumok követésére. A tanító adatok 5 különböző helyszín forgalmát tartalmazzák. minden helyszín más tulajdonságokkal bír, ezért nem lehet generalizálni a tanítási folyamatot, nem lehet egy univerzális

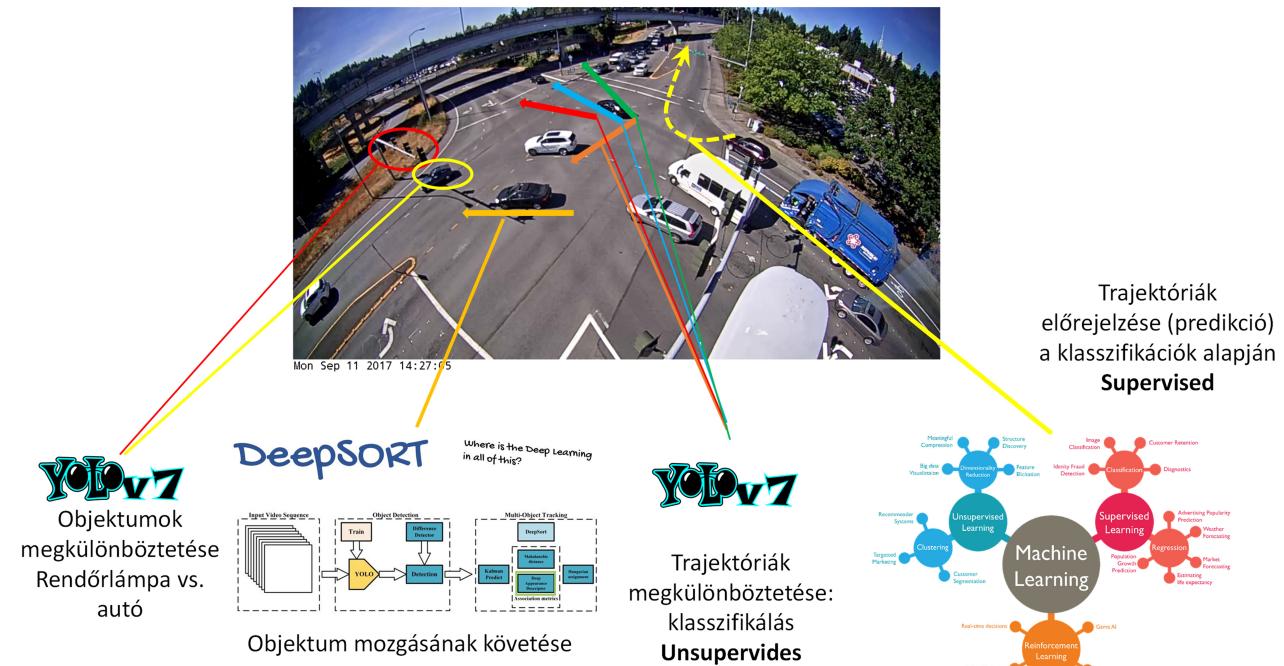


Fig. 1. Bellevue Newport kereszteződés



Fig. 2. Bellevue Eastgate kereszteződés

modellt betanítani ami minden közlekedési helyszínre alkalmazható egyaránt. 4 videót Bellevue város github oldaláról gyűjtöttük, amiknek az elérhetőségét függelékként csatoljuk, a kereszteződések 1. 2. 3. 4. képeken láthatók, az ötödik videó La Grange-ból származik lássd 5. A videók pontos elérhetőségét a mellékletben található *urls.txt*-ben adtuk meg.

## 2 KAPCSOLÓDÓ KUTATÁSOK

Sok ITS-el kapcsolatos kutatásban tárgyalják a forgalom folyás (traffic flow) előrejelzését. [Paul et al. 2017] összehasonlítja az eddig kutatott és használt modellek, mint például Kalman Filtering, k-nearest neighbor (k-NN), mesterséges neurális hálók, stb., pontosságát és sebességét, ezen modellek továbbkutatását, mivel egyre növekednek a különböző szenzorok által begyűjtött traffic flow adatok, így



Fig. 3. Bellevue NE kereszteződés



Fig. 4. Bellevue SE kereszteződés



Fig. 5. La Grange KY North

ez a terület belépett a *Big Data* korszakába. [Rossi et al. 2021] is a traffic flow előrejelzését és generálását tárgyalja, Floating Car Data (FCD) adathalmazokon betanított, Hosszú-Rövid-Távú memóriájú és Generatív versengő hálókkal.

## 2.1 YOLO

YOLO (You Only Look Once) egy nagyon hatékony objektumdetektáló algoritmus, amely képes nagyon gyorsan észlelni és besorolni az objektumokat egy képen vagy videón. Az YOLO algoritmus működése a következő lépésekkel áll:

- (1) Bemeneti kép előkészítése: A kép előkészítése magában foglalja a normalizálást és a méretarányhoz való igazítást annak érdekében, hogy az YOLO algoritmus hatékonyan dolgozhasson a képpel.
- (2) Vektor előállítása: Az YOLO algoritmus a bemeneti képet a vektorizálás segítségével elemzi, amelynek eredménye egy tensor lesz, amely az objektumok lokalizációjához és azok osztályozásához szükséges információkat tartalmazza.
- (3) Konvolúciós hálózat alkalmazása: Az YOLO algoritmus egy kiterjedt konvolúciós hálózatot alkalmaz a vektorra, amelynek célja az objektumok lokalizálása és azok osztályozása.

- (4) Konvolúciós hálózat alkalmazása: Az YOLO algoritmus egy kiterjedt konvolúciós hálózatot alkalmaz a vektorra, amelynek célja az objektumok lokalizálása és azok osztályozása.
- (5) Objektum lokalizálása és osztályozása: Az YOLO algoritmus az általa előállított tensoron keresztül végzi az objektumok lokalizálását és azok osztályozását. Az algoritmus meghatározza az objektumok koordinátáit és a hozzájuk tartozó osztályt.

Az YOLO algoritmus előnye, hogy nagyon gyors és hatékonyan kezeli az objektumok lokalizálását és azok osztályozását. Az algoritmus gyakran jobb teljesítményt nyújt, mint a hasonló módszerek, és a különböző objektumokat a bemeneti képen gyorsan és hatékonyan azonosítja. Azonban az YOLO algoritmus hibázhat, ha az objektumok nagyon hasonlóak egymáshoz vagy a háttérhez, és nagyobb hibát eredményezhet, ha az objektumok nagyon kicsik a képen. Legfrissebb változata a Yolov7 felülműlja sebességeben és pontosságban a modern konvolúciós hálókat (lásd 6 7 8). Beágyazott rendszerekben és videókártyákon is egyaránt jó a teljesítménye, ezért az ITS területén alkalmazható.

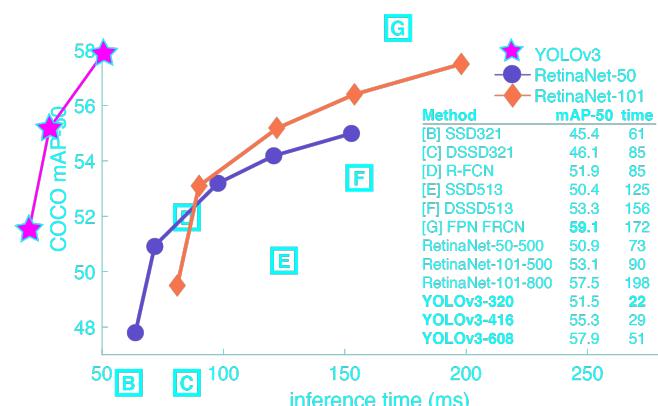


Fig. 6. YOLOv3 Performance

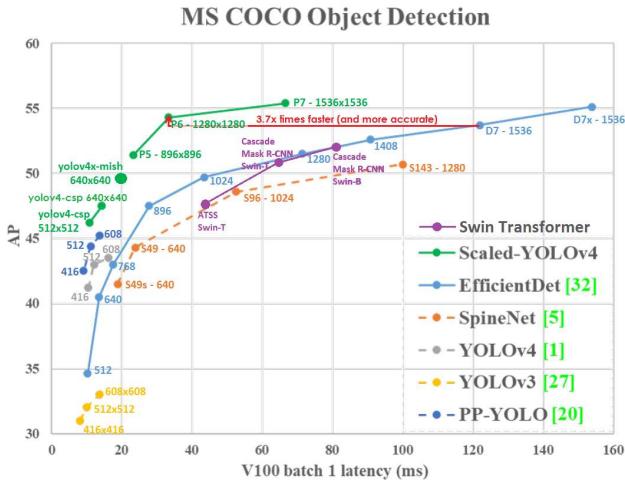


Fig. 7. YOLOv4 Performance

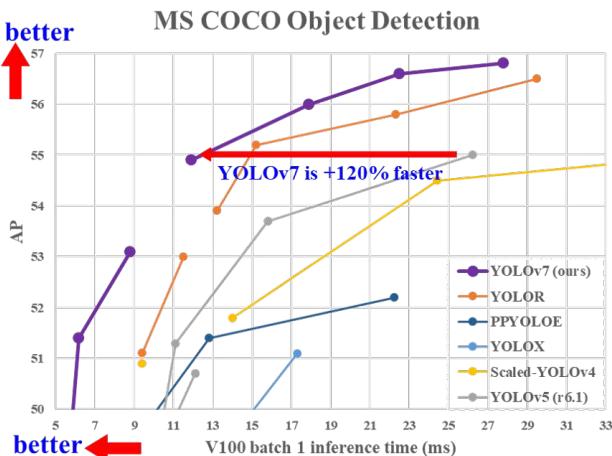


Fig. 8. YOLOv7 Performance

## 2.2 DeepSORT

Az YOLO algoritmus előnye, hogy nagyon gyors és hatékonyan kezeli az objektumok lokalizálását és azok osztályozását. Az algoritmus gyakran jobb teljesítményt nyújt, mint a hasonló módszerek, és a különböző objektumokat a bemeneti képen gyorsan és hatékonyan azonosítja. Azonban az YOLO algoritmus hibázhat, ha az objektumok nagyon hasonlóak egymáshoz vagy a háttérhez, és nagyobb hibát eredményezhet, ha az objektumok nagyon kicsik a képen. A DeepSORT algoritmus működése a következő lépésekkel áll:

- 1) Objektumdetektálás: Az algoritmus először objektumdetektálással azonosítja az összes objektumot a videofelvételen, például a YOLO objektumdetektáló algoritmust használva.

## DeepSORT

Where is the Deep Learning in all of this?

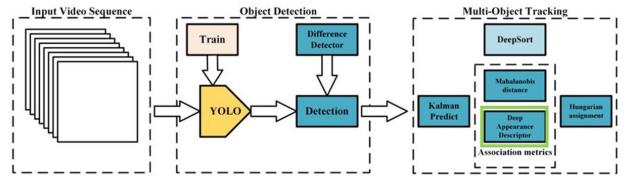


Fig. 9. Objektum követés

- 2) Jellemzők kinyerése: A DeepSORT az objektumok jellemzőit (pl. méret, sebesség, szín) kinyeri, hogy a következő lépésben a következő objektumot azonosítani tudja.
- 3) Objektumazonosítás: Az algoritmus használ egy "tracklet" nevű algoritmust, hogy azonosítja és kövesse az objektumokat az időben. A "tracklet" az objektum jellemzőit használja, hogy azonosítja az adott objektumot a videofelvétel további részein.
- 4) Címkezés: Az objektumokat azonosítják egyedi azonosítókkal, hogy az algoritmus megkülönböztethesse azokat az egyes videofelvételeken.
- 5) Korszakosítás: A DeepSORT algoritmus általánosan a Kalman-szűrőt használja, amely folyamatosan frissíti az objektumok helyzetének becslését. A Kalman-szűrő segít az algoritmusnak megjósolni az objektumok további helyzetét a videofelvétel során.

A DeepSORT algoritmus előnye, hogy nagyon stabil és pontos objektumkövetést biztosít akkor is, ha az objektumok átmennek más objektumok mögött vagy ha azok mozgása elég bonyolult. Az algoritmus nagyobb pontosságot nyújt a hagyományos objektumkövetési algoritmusokhoz képest, és képes megbirkózni a nagy sebességű objektumok követésével is. Azonban az algoritmus nagyobb számítási erőforrásokat igényel, és magasabb szintű számítási készséget igényel az implementáláshoz.

## 3 ADATHALMAZOK KIALAKÍTÁSA

A kutatás során saját adathalmazok kialakítására volt szükség. Az adatok begyűjtésére és eltárolására saját alkalmazást és keretrendszert fejlesztettünk ki. A szoftver keretrendszert python nyelven írtuk meg, a forráskód ezen a linken megtalálható [http://github.com/Pecneb/computer\\_vision\\_research](http://github.com/Pecneb/computer_vision_research). A fejlesztés során a következő programkönyvtárakat használtuk OpenCV [Bradski 2000], Numpy [Harris et al. 2020], Pandas [pandas development team 2020], Scikit-Learn [Pedregosa et al. 2011], Matplotlib [Hunter 2007], SQLite [Hipp 2020], Joblib [Joblib Development Team 2020]. Az adathalmazokat SQLite adatbázisban és joblib fájlokban

tároltuk el. Azért döntöttünk így, hogy két féle módon is eltároljuk az adathalmazokat, mert az SQL adatbázist univerzálisan bármilyen adatbáziskezelővel, vagy más programnyelvvel be lehet olvasni, viszont a joblib fájlok sokkal gyorsabban be lehet töltetni pythonnal.

### 3.1 Adatstruktúra

Az adatstruktúrát SQL schema-ként, és python osztály-ként is definiáltuk.

```
CREATE TABLE IF NOT EXISTS objects (
    objID INTEGER PRIMARY KEY NOT NULL,
    label TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS detections (
    objID INTEGER NOT NULL,
    frameNum INTEGER NOT NULL,
    confidence REAL NOT NULL,
    x REAL NOT NULL,
    y REAL NOT NULL,
    width REAL NOT NULL,
    height REAL NOT NULL,
    vx REAL NOT NULL,
    vy REAL NOT NULL,
    ax REAL NOT NULL,
    ay REAL NOT NULL,
    vx_c REAL NOT NULL,
    vy_c REAL NOT NULL,
    ax_c REAL NOT NULL,
    ay_c REAL NOT NULL,
    FOREIGN KEY(objID) REFERENCES objects(objID)
);
CREATE TABLE IF NOT EXISTS metadata (
    historyDepth INTEGER NOT NULL,
    yoloVersion TEXT NOT NULL,
    device TEXT NOT NULL,
    imgsize INTEGER NOT NULL,
    stride INTEGER NOT NULL,
    confidence_threshold REAL NOT NULL,
    iou_threshold REAL NOT NULL,
    k_velocity REAL NOT NULL,
    k_acceleration REAL NOT NULL
);
```

Minden követett objektum egyedi azonosítóval lett ellátva. Az objektumhoz tartozó detektálások külön táblába lett kiszervezve, ahol az *objID* idegen kulcsal kapcsoljuk az *objektumok* táblához. Egy objektumhoz az egyedi azonosítón kívül tartozik egy *label* amit a YOLO objektum detektálótól kap, ez lehet pl. autó, személy, teherautó, stb. Az objektumokhoz tartozó detektálások tartalmazzák a képkocka számát, amikor a detektálás történt, a konfidenciát, hogy mennyire biztos az objektumfelismerő a hozzárendelt *label*-ben, az objektum *x*, *y* kordinátáját, az objektum szélességét *width* és magasságát *width*, sebességét *v<sub>x</sub>*, *v<sub>y</sub>* és gyorsulását *a<sub>x</sub>*, *a<sub>y</sub>*, amik

a deepSORT által kalkulált értékek, így még külön a koordinátákból kiszámolt *v<sub>x<sub>c</sub></sub>*, *v<sub>y<sub>c</sub></sub>* sebességet és *a<sub>x<sub>c</sub></sub>*, *a<sub>y<sub>c</sub></sub>* gyorsulást is eltároltuk. Ezek mellett még a konfigurációs adatokat is külön táblában tároljuk, hogy később meg lehessen ismételni a detektálást. A koordinátákat a videó méretének megfelelően leskálázzuk 0 - 1 értékek köré. Ha a videó kép szélesség *w*, magasság *h*, akkor a képarány *r* =  $\frac{w}{h}$ , és az eltárolt koordináták

$$x = \frac{x_0}{w} * r, y = \frac{y_0}{w} * r \quad (1)$$

$$v_x = \frac{v_{x0}}{w} * r, v_y = \frac{v_{y0}}{w} * r \quad (2)$$

$$a_x = \frac{a_{x0}}{w} * r, a_y = \frac{a_{y0}}{w} * r \quad (3)$$

$$v_{x_c} = \frac{v_{xc0}}{w} * r, v_{y_c} = \frac{v_{yc0}}{w} * r \quad (4)$$

$$a_{x_c} = \frac{a_{xc0}}{w} * r, a_{y_c} = \frac{a_{yc0}}{w} * r \quad (5)$$

### 3.2 Objektumdetektálás

Az objektumdetektáláshoz a fennt említett YOLO modellt használtuk. Kutatásunk kezdetekor, a YOLO 4-es verziójával kezdtük dolgozni, de később átváltottunk a jobb pontosságot és sebességet igérő 7-es verzióra.

**3.2.1 YOLOv4.** YOLO 4-es verzióját, C-ben implementálták. Hogy fel tudjuk használni, írnunk kellett egy python API-t, ami meg tudtunk hívni a deketáló programunkban.

**3.2.2 YOLOv7.** A YOLOv7 viszont már python-ban implementálták amihez már sokkal könnyebb volt API-t programozni és használni. Emellett, gyorsaságban és pontosságban is felülmúltja a 4-es verziót (lásd 8).

### 3.3 Objektumkövetés

Ahhoz, hogy trajektóriák alapján tudjunk szabájosságokat felismerni a forgalomban, pontos objektumkövetésre volt szükségünk. Eleinte saját objektumkövető algoritmust használtunk, ami deketálások euklideszi távolsága alapján próbálta meg követni az objektumokat. Ezzel az volt a gond, hogy hosszabb kitakarás után nem találta meg az objektumot, így egy új objektumnak számított, ami a kép közepéből bukkant fel. Ennek a problémának a kiküszöbölésére próbáltuk ki a DeepSORT algoritmust.

**3.3.1 DeepSORT.** A DeepSORT algoritmus pythonban implementált változatát integráltuk a mi programunkba.

## 4 KLASZTEREZÉS

A klaszterezés segítségével lehet az adathalmazból előállítani a klasszifikáció alapjául szolgáló klasszokat. Ahhoz, hogy az a rengeteg trajektóriából és detektálásból számunkra felhasználható információ keletkezzen, meg kell határoznunk feature vectorokat, amik a trajektóriákra jellemző értékeket tartalmaznak. Ebben a feature téren fogja a klaszterező algoritmus megtalálni az egymáshoz közeli, hasonló trajektóriákat.

#### 4.1 Adattisztítás

A klaszterezés előtt a nyers adatokat fel kell dolgoznunk, hogy az esetleges hibás, zajos detektálások, trajektóriák miatt kapjunk fals klasztereket. Az objektum detektálás és követés nem tökéletes, rossz fényviszonyok, hosszabb eltakarások miatt a trajektóriák megszakadhatnak, ezért ki kell választani az egyben maradt trajektóriákat. Három szűrő algoritmust futtattunk az adathalmazon. Elsőnek a trajektóriák belépő és kilépő pontjainak az euklideszi távolsága alapján szűrtünk. Majd a kép széleit meghatározzuk min max kiválasztással, és azokat a trajektórákat választjuk ki amiknek a szélektől meghatározott távolságra vannak a belépő és kilépő pontjaik.

**4.1.1 DeepSORT pontatlanság.** Kutatásunk során azt tapasztaltuk, hogy a DeepSORT és a YOLO pontatlanságai felerősítik egymást. A YOLO hajlamos néha táblákat vagy rendőr lám-pákat autóknak nézni, és ekkor a DeepSORT is elkezdi követni. Egy olyan hibáját is felfedeztük a DeepSORT-nak, hogy egy objektumról áttapad a követés egy másik objektumra, ami fals trajektóriákat hoz létre. A DeepSORT-nak lehet finomhangolni a paramétereit, ami nem bizonyult akkora javulásnak, ezért útőlagos szűréssel kellett korrigálnunk ezt a hibát. Az algoritmus végig iterál a trajektóriák pontjain és kiszámítja az egymást követő detektálások euklideszi távoláságát, és ha egy küszöbérték felett vannak akkor eldobjuk a trajektóriát. A következő képeken láthatók a klaszterek szűrés előtt és után (lásd 10 11).

#### 4.2 Feature vektorok

Klaszterezéshez 4 és 6 dimenziós feature vektorokat használtunk. A 6 dimenziós vektorokat a DeepSORT hibájának a kiszűrésére hoztuk létre, felépítésük a következő [belépő x,y középső x,y kilépő x,y], de a kifejlesztett szűrő hatékonyabbnak bizonyult, és a kevesebb dimenzió is előnyt jelent, ezért maradtunk a 4 dimenziós feature vektor mellett, aminek a felépítése: [belépő x,y kilépő x,y].

#### 4.3 Klaszterezési algoritmusok

Klaszterezéshez több fajta algoritmust teszteltünk. A legjobb eredményeket az OPTICS (Ordering Points To Identify the Clustering Structure) [Ankerst et al. 1999] algoritmus adta. Aminek az eredményei fenti képeken láthatók (lásd. 10). OPTICS-on kívül leteszteltük a KMeans, DBSCAN és BIRCH algoritmusokat.

**KMeans.** A KMeans klaszterezés egy unsupervised machine learning algoritmus, amely célja az adatok csoportosítása oly módon, hogy azonos klaszterbe tartozó adatok közötti távolság minimális legyen, míg az eltérő klaszterek közötti távolság maximális. Az algoritmus működése a következő lépésekből áll:

- (1) Centroidok inicializálása: Az algoritmus véletlenszerűen inicializál k centroidet a dataseten, ahol k a klaszterek száma.
- (2) Adatok csoportosítása: Az algoritmus minden adatponthoz hozzárendeli a legközelebbi centroidet, és azonos

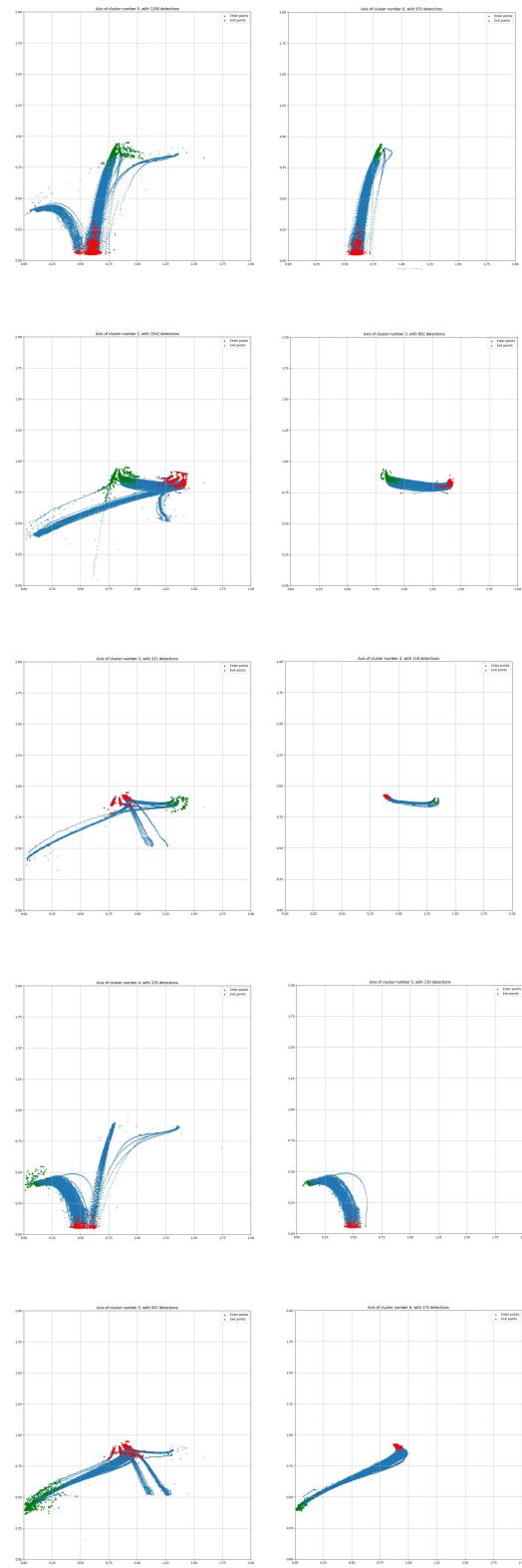


Fig. 10. Klaszterezés szűrő  
, Vol. 1, No. 1, Article . Publication date: April 2023.

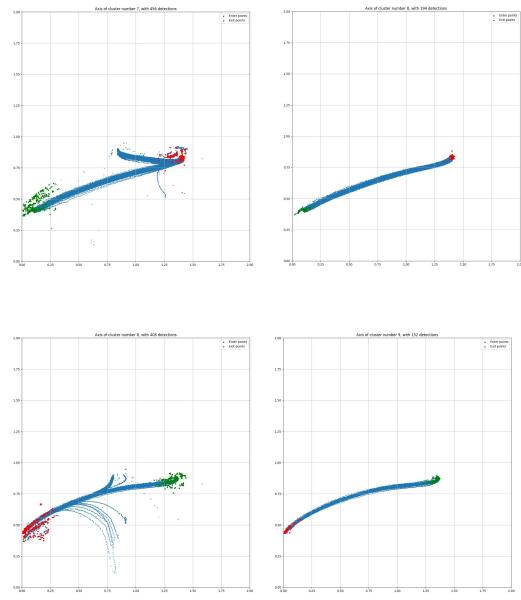


Fig. 11. Klaszterezés szűrő

klaszterbe helyezi azokat az adatpontokat, amelyeknek a centroidja megegyezik.

- (3) Centroidok újraszámolása: Az algoritmus újraszámolja a centroidok pozíciót az adatok csoportosítása után, hogy azok a klaszterben található adatpontok átlagértékének megfelelően helyezkedjenek el.
- (4) Lépések ismétlése: Az algoritmus addig ismételgeti a 2. és 3. lépéseket, amíg az adatpontok klaszterezése konvergens állapotba nem jut, azaz az adatpontok csoportosítása már nem változik, vagy az algoritmus előre meghatározott maximális iterációs számhoz ér.
- (5) Klaszterek értékelése: Az algoritmus kiértékeli a klaszterek minőségét, például a csoportokban lévő adatpontok közötti távolságot, és dönti el, hogy a csoportokat újra kell-e szervezni.

A KMeans klaszterezés előnye, hogy egyszerű és gyors algoritmus, amely hatékonyan használható az adatok csoportosítására. A klaszterek számát könnyen meg lehet adni, és az algoritmus gyorsan konvergál. Azonban az algoritmus érzékeny az inicializálási folyamatokra, és gyakran találhatóak olyan csoportok, amelyek nem teljesen homogének. Emellett KMeans  $n\_clusters$  - klaszterek száma - paraméterét előre kell definiálni, aminek meghatározására próbálkoztunk különböző metrikákat felhasználni, hogy automatizálható legyen a klaszterezési lépés. Ezek a metrikák a Silhouette Coefficient [Rousseeuw 1987], Calinski-Harabasz Index [Caliński and JA 1974] és Davies-Bouldin Index [Davies and Bouldin 1979]. Elbow diagramok segítségével próbáltuk eldönteni, hogy milyen értéket érdemes adni az  $n\_clusters$  paraméternek lásd 12. A mérőszámok konzisztenlesen alacsony értéket adtak, egy

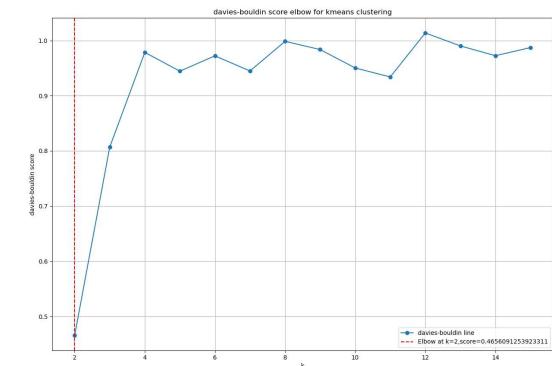
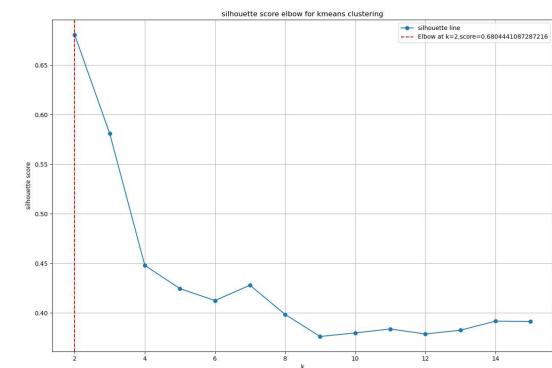
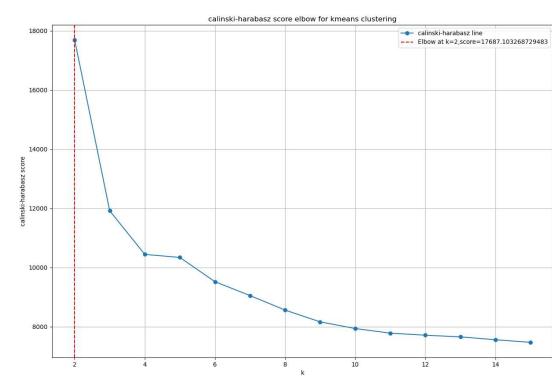


Fig. 12. Elbow diagramok

négy ágú kereszteződésnél, ahol jóval több klaszterbe sorolhatók a trajektoriák. A KMeans használatát ezért elvetettük.

A *BIRCH* (*Balanced Iterative Reducing and Clustering using Hierarchies*), egy gyors és hatékony hierarchikus klaszterezési algoritmus, amelyet nagy mennyiségű adat gyors csoportosítására fejlesztettek ki. Az algoritmus célja, hogy az adatokat összesítse a memóriában, és a klaszterek készítése

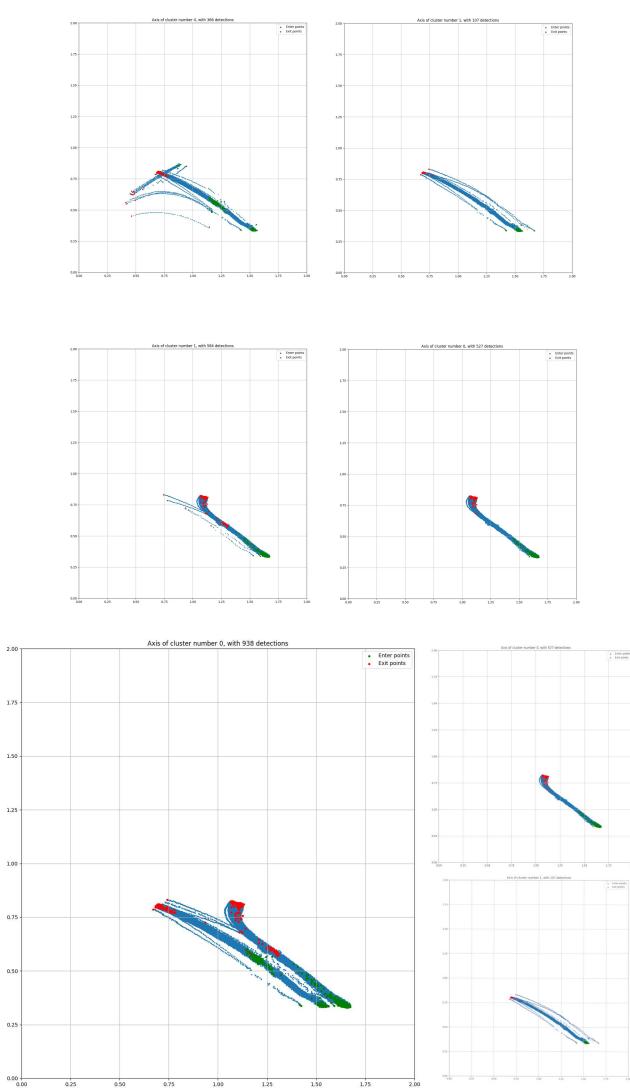


Fig. 13. KMeans, BIRCH, DBSCAN vs OPTICS

korán ne kelljen minden adatpontot az egész adathalmazon végigvinni. Az algoritmus a következő lépésekkel áll:

- (1) Adatok aggregálása: Az adatok aggregálása során az algoritmus egymás mellé helyezi az adatokat az összetartozó klaszterekben. Az aggregálási folyamat során az algoritmus az adatokat kisebb csoportokba osztja, és azokat összevonja egy aggregált reprezentációba.
- (2) Hierarchikus csoportosítás: Az algoritmus létrehozza az aggregált adathalmaz hierarchikus reprezentációját. Az adatokat egy fa szerkezetben helyezi el, ahol a gyökér a teljes adathalmaz, a levél pedig az egyes adatpontokat tartalmazza.
- (3) Clustering: Az algoritmus elvégzi az adatok klaszterezését a hierarchikus fa struktúra alapján. A

klaszterek létrehozása iteratív folyamat, amelyben az algoritmus egymás után dolgozza fel a fa szintjeit. Az algoritmus minden szinten klaszterezést végez, és az előző szinten megtalált klasztereket használja a következő szinten végzett csoportosításhoz.

A BIRCH algoritmus előnye, hogy hatékonyan kezeli a nagy mennyiségi adatokat, és minimális memóriahasználatot igényel. Az algoritmus gyorsan fut, és lehetővé teszi a csoportok hierarchikus struktúrájának vizsgálatát. Azonban az algoritmus nem alkalmas olyan adatokra, amelyeket nehéz aggregálni, és az adatok aggregálása során elveszhetnek a finom részletek. A futtatott tesztek alapján elmondható, hogy a BIRCH algoritmus sokszor egybevon klaszter beállításokat vagy kimeneteket, ami miatt több más irányból jövő, vagy több más irányba kilépő objektumokat sorol azonos klaszterekbe. *threshold* paraméterrel lehet a klaszterek méretét szabályozni, amivel javítható az egybevonott klaszterek száma, a kutatás során futtatott tesztek alapján, még így is az OPTICS adta a legtisztább klasztereket.

*DBSCAN (Density-Based Spatial Clustering of Applications with Noise)*, egy hatékony klaszterezési algoritmus, amely a sűrűség alapján klaszterez. Az algoritmus célja, hogy megtalálja a sűrűségi alapú klasztereket az adathalmazban, és az adatpontokat azonosítja, amelyek nem tartoznak egyik klaszterhez sem, az úgynevezett zajokat. Az algoritmus három fő paramétere a klaszterek sűrűségének küszöbe *eps*, az adatpontok minimum szomszédjainak száma *min\_samples* és az adatpontok kiindulási pozíciója. Az algoritmus lépései a következők:

- (1) Választ véletlenszerűen egy adatpontot, amely még nem lett klaszterezve.
- (2) Találja meg az összes adatpontot, amelyekre az *eps* sugarú kör középpontjából el lehet jutni.
- (3) Ha az adatpontok száma nagyobb, mint a *min\_samples*, akkor létrehoz egy új klasztert és hozzáadja az összes adatpontot a klaszterhez. Ha az adatpontok száma kisebb, mint a *min\_samples*, megjelöli az adatpontot zajként.
- (4) Folyamat megismétlése az összes nem klaszterezett adatponttal.

*Az OPTICS (Ordering Points To Identify the Clustering Structure)*, egy másik klaszterező algoritmus, amely a sűrűség alapú klaszterezést használja. Az algoritmus a DBSCAN-hoz hasonlóan az adatpontok közötti sűrűségi kapcsolatokat használja a klaszterek meghatározásához, de az OPTICS további információt szolgáltat az adathalmaz klaszterezett struktúrájáról. Az algoritmus az adatpontok távolságát és azok sűrűségét is figyelembe veszi a klaszterek meghatározásához. Az OPTICS algoritmus lépései a következők:

- (1) Válasszunk ki egy véletlenszerű adatpontot, amely még nem került klaszterezésre.
- (2) Megkeressük az összes szomszédos adatpontot, és kiszámítjuk a távolságát az adott adatponttól.

- (3) A szomszédos adatpontokat távolság és sűrűség szerint rendnezi.
- (4) Létrehoz egy "optikai" sorrendet, amelyben az adatpontokat rendezzük a távolságuk és a sűrűségük szerint. Ez lehetővé teszi, hogy az algoritmus később könnyebben megtalálja a klasztereket és a zajokat.
- (5) Ha az adatpontot egy klaszterhez lehet rendelni, akkor adjuk hozzá a klaszterhez.
- (6) A folyamatot megismétli az összes nem klaszterezett adatpontra.

Az OPTICS algoritmus előnye, hogy lehetővé teszi a klaszterek és a zajok meghatározását egyaránt, és további információkat is szolgáltat az adathalmaz klaszterezett struktúrájáról, mint például a klaszterek hierarchiájáról és a klaszterek közötti távolságról. Az OPTICS azonban az adathalmazok nagy méretű és magas dimenziós esetében nagyon lassú lehet, és nagy erőforrásigényt igényelhet a klaszterezéshez.

Paraméterezésben a DBSCAN annyiban különbözik az OPTICS-tól, hogy *max\_eps* paraméter helyett, ami egy távolság tartományt ad meg, az *eps* paramétert használja, ami pontos távolságot ad meg.

A 13. képen, a KMeans, BIRCH, DBSCAN klasztereket állítjuk szembe az OPTICS által rendezett klaszterekkel. Látható, hogy az OPTICS, nagyon hatékonyan tudta kiszűrni a zajos trajektóriákat, és nem vont egybe kimeneti vagy beállítási klasztereket.

#### 4.4 Paraméterek kiválasztása

A megfelelő paraméterek kiválasztása a klaszterezéshez igen fontosnak bizonyult. Ezt a gyűjtött adathalmazokon kézzel kellett finomhangolnunk. A halmazok minimum számosságát a *min\_samples* paraméterrel lehet szabályozni, a pontok egymástól való távolságának felső határát *max\_eps*-el lehet megadni. Az távolság kiszámítására használt methódust *metric*-el lehet megadni. A *xi* paraméterrel az elérési plot minimum meredekségét lehet megadni. ami a klaszterek határát szabja meg. Az adathalmazra alkalmazható megfelelő paramétereket nem tudtuk generalizálni, kézzel kellett finomhangolnunk. A plotokon látható klaszterek megtalálásához *min\_samples* = 50, *max\_eps* = 0.1, *metric* = 'minkowski' és *xi* = 0.15 paramétereket használtunk.

### 5 KLASSZIFIKÁCIÓ

#### 5.1 Multiclass

A több osztályos klasszifikálás egy olyan feladat, amikor több mint 2 osztály van, és minden feature vektor csak egy osztályba tartozhat. Az alapvető megközelítés a többosztályos klasszifikációra az, hogy a modell tanítása során az összes lehetséges kategóriát együttesen kell figyelembe venni. Ez azt jelenti, hogy minden egyes kategóriát egy külön osztályként kell kezelni, és a modell tanulásakor figyelembe kell venni az összes osztályt. Az osztályozó modell célja, hogy az adathalmazból kiválasztott jellemzők és az osztályok közötti kapcsolatok alapján olyan döntési fát vagy osztályozó algoritmust

hozzon létre, amely képes az új adatok osztályozására. A multiclass klasszifikációhoz különböző algoritmusok használhatók, például a Random Forest, Support Vector Machine (SVM), k-Nearest Neighbor (kNN), Decision Tree és Deep Neural Network (DNN). A megfelelő algoritmus kiválasztása az adathalmaz méretétől, dimenziójától, a célkitűzésekkel és az adatok jellegétől függ. A Multiclass klasszifikáció kevesebb osztály számnál jó eredményt adhat, de a mi esetünkben ahol 10-15 osztály is lehet, ami azt jelenti, hogy nem lehet elérni nagy pontosságot. Ennyi osztály közül nehéz pontosan elatlálni melyik osztályba tartozik egy trajektória.

#### 5.2 Binary

A binary (kétosztályos) klasszifikáció egy olyan gépi tanulási probléma, amelyben az adathalmazban szereplő objektumokat vagy eseményeket két kategóriába kell osztályozni. Például megkülönböztethetjük a "spam" és "nem spam" leveleket, vagy az "egészséges" és "beteg" betegeket az orvosi diagnózisban. Az alapvető megközelítés a binary klasszifikációra az, hogy az osztályozó modell olyan döntési határt hoz létre az adathalmazban található adatok és az osztályok között, amely megkülönbözteti az egyik kategóriába tartozó adatokat a másiktól. Ennek az eredménye egy bináris predikció, amely azt jelzi, hogy egy adott adat az egyik vagy a másik kategóriába tartozik.

#### 5.3 OneVsRest

A One-vs-Rest (OvR), más néven One-vs-All (OvA) klasszifikáció egy olyan többosztályos osztályozási technika, amelynek célja, hogy különböző osztályok között megkülönböztetést végezzen. Az OvR-ben a különböző osztályok közötti különbségeket az egyik osztályhoz képest határozzák meg. Ezt az osztályt "egy" osztálynak nevezik, és a többi osztályt "a többi" osztályoknak. Az OvR algoritmusban egy osztályozó modellt hoznak létre minden egyes osztály és a többi osztályok közötti megkülönböztetésre. Ez azt jelenti, hogy ha van például 5 osztályunk, akkor 5 különböző bináris klasszifikátorra van szükségünk, amelyek mindegyike egy adott osztályt különböztet meg a többi osztálytól. A bináris osztályozók által létrehozott modellt használják az osztályozásra. Az osztályozó modellnek két kimenete van, "1" vagy "0". Ha a modell kimenete "1", akkor az adott minta az adott osztályhoz tartozik, ha a kimenete "0", akkor az adott minta nem tartozik az osztályhoz. Az OvR osztályozó előnye, hogy egyszerűen használható, mivel csak bináris osztályozókat kell alkalmazni minden egyes osztályra, és használható, ha az osztályok közötti határok nincsenek jó meghatározva.

#### 5.4 Machine Learning modellek

Kutatásunk során több féle machine learning modellt teszteltünk: KNN (KNearestNeighbors), GNB (GaussianNaiveBayes), MLP (MultiLayerPerceptron), SGD (StochasticGradientDescent), SVM/SVC (SupportVectorMachine/SupportVectorClassifier) [Chang and Lin 2011], DT (DecisionTree) [Breiman et al. 1984]. Ezekből a GNB,

| Bellevue Newport   |          |        |        |
|--------------------|----------|--------|--------|
| Metrics            | Balanced | Top 1  | Top 2  |
| KNN                | 90.57%   | 95.50% | 99.12% |
| GNB                | 63.92%   | 73.25% | 90.10% |
| MLP                | 58.49%   | 81.93% | 89.43% |
| SGD Modified Huber | 45.43%   | 66.39% | 83.54% |
| SGD Log Loss       | 40.41%   | 60.70% | 79.69% |
| SVM                | 77.87%   | 89.29% | 96.61% |
| DT                 | 90.95%   | 93.64% | 95.06% |

Table 1. Bellevue Newport Feature Vektor V1

| Average Accuracy Feature Vector v1 |          |        |        |
|------------------------------------|----------|--------|--------|
| Metrics                            | Balanced | Top 1  | Top 2  |
| KNN                                | 94.16%   | 96.71% | 99.46% |
| SVM                                | 81.65%   | 90.82% | 98.05% |
| DT                                 | 93.11%   | 94.88% | 96.03% |

Table 2. Testset Feature Vektor V1

| Average Accuracy Feature Vector v7 |          |        |        |
|------------------------------------|----------|--------|--------|
| Metrics                            | Balanced | Top 1  | Top 2  |
| KNN                                | 92.08%   | 95.61% | 98.66% |
| SVM                                | 88.72%   | 93.86% | 98.92% |
| DT                                 | 89.46%   | 93.30% | 94.55% |

Table 3. Testset Feature Vektor V7 Stride 15

| Average Accuracy Feature Vector v7 stride 30 |          |        |        |
|--|----------|--------|--------|
| Metrics                                      | Balanced | Top 1  | Top 2  |
| KNN  | 92.68%   | 95.96% | 98.79% |
| SVM  | 88.67%   | 93.49% | 98.91% |
| DT   | 89.87%   | 93.17% | 94.53% |

Table 4. Testset Feature Vektor V7 Stride 30

MLP és SGD nem adott jó eredményeket ami látható az alábbi táblázatban 1, az eredmények megismétlődtek későbbi tesztekben, ezért ezeket a modelleket nem tárgyaljuk. A legjobb eredményeket a KNN adta minden esetben 90% felett teljesített. A második legjobb a DecisionTree lett, ami átlagban balanced accuracy-ban az SVM felett teljesített, és Top 1 accuracyban is csak tizedekkel maradt le a 7. feature vektor használatakor, az 1. verzióval 4%-al jobban teljesített. A mérések eredményei a 2. és 3. táblázatban láthatók.

A *K-Nearest Neighbors (KNN)* egy egyszerű és hatékony osztályozó algoritmus, amely az adatok közötti távolság alapján osztályozza a bemeneti adatokat. Az algoritmus lényege, hogy egy adott bemeneti adathoz hasonló adatokat keres a tanuló adathalmazból, majd azok osztálycíméit megnézve meghatározza az adat osztályát. Az algoritmus

| Cross-Validation Average Accuracy Feature Vector v1 |          |        |        |
|---|----------|--------|--------|
| Metrics   | Balanced | Top 1  | Top 2  |
| KNN   | 92.60%   | 96.06% | 99.38% |
| SVM   | 81.21%   | 89.76% | 97.79% |
| DT  | 92.43%   | 94.55% | 95.97% |

| Cross-Validation Average Accuracy Feature Vector v7 |          |        |        |
|---|----------|--------|--------|
| Metrics   | Balanced | Top 1  | Top 2  |
| KNN   | 90.74%   | 95.49% | 98.39% |
| SVM   | 87.36%   | 94.03% | 98.85% |
| DT  | 89.12%   | 93.56% | 94.95% |

Table 5. Cross-Validation Feature Vektor V1

| Cross-Validation Average Accuracy Feature Vector v7 stride 15 |          |        |        |
|---|----------|--------|--------|
| Metrics   | Balanced | Top 1  | Top 2  |
| KNN   | 91.15%   | 95.64% | 98.54% |
| SVM   | 87.28%   | 93.69% | 98.60% |
| DT  | 89.55%   | 93.73% | 95.15% |

Table 6. Cross-Validation Feature Vektor V7 Stride 15

| Cross-Validation Average Accuracy Feature Vector v7 stride 30 |          |        |        |
|---|----------|--------|--------|
| Metrics   | Balanced | Top 1  | Top 2  |
| KNN   | 91.15%   | 95.64% | 98.54% |
| SVM   | 87.28%   | 93.69% | 98.60% |
| DT  | 89.55%   | 93.73% | 95.15% |

Table 7. Cross-Validation Feature Vektor V7 Stride 30

először szükséges, hogy az adatokat előkészítse. Ez magában foglalhatja az adatok normalizálását, standardizálását, az outlier-ek kezelését, valamint a kategorikus adatok konvertálását numerikus formába. Az algoritmus működése a következő lépésekkel áll:

- (1) Távolságok számítása: Az algoritmus először számítja ki az összes tanuló adatpont és a bemeneti adatpont közötti távolságot. A leggyakrabban használt távolságmértékek az euklideszi és manhattan távolságok.
- (2) K legközelebbi szomszéd kiválasztása: Az algoritmus a távolságok alapján kiválasztja a K legközelebbi szomszédot a bemeneti adatpontnak. A K értéke általában egy páros szám, hogy elkerüljük a döntések holtversenyét.
- (3) Döntés meghozása: Az algoritmus a K legközelebbi szomszéd osztálycíméinek többségi szavazatával dönti el, hogy melyik osztályba sorolja a bemeneti adatpontot.

Az előnye, hogy a KNN algoritmus könnyen értelmezhető és egyszerűen használható. Az algoritmus jól működik a kisebb méretű adathalmazokon, különösen akkor, ha az adatok egyszerű struktúrával rendelkeznek. A KNN továbbá jól alkalmazható olyan feladatokra, ahol a határok nem lineárisak, és ahol a szokásos statisztikai módszerek nem elegendőek. Az algoritmusnak azonban vannak korlátai, például az, hogy az algoritmus futási ideje növekszik az adathalmaz méretével,

valamint hogy az eredmények érzékenyek lehetnek az adatpontok elhelyezkedésére.

Az *SVM* (*Support Vector Machine*). egy erőteljes, nem-lineáris osztályozó algoritmus, amelynek célja egy határvonal (vagy hipersík) megtalálása az adatok között. Az algoritmus a bemeneti adatokat olyan módon osztályozza, hogy a döntési határvonal az osztályok közötti legnagyobb távolságot biztosítsa. Ez a távolság a két legközelebbi adatpont közötti távolság, és a margin-nek nevezik. Az SVM algoritmus működése a következő lépésekkel áll:

- (1) Adatok előkészítése: Az adatokat előkészítjük, eltávolítjuk a hiányzó adatokat, valamint normalizáljuk vagy standardizáljuk az adatokat a hatékonyabb tanulási folyamat érdekében.
- (2) Határvonal (hipersík) keresése: Az SVM algoritmus az adatokat olyan módon osztályozza, hogy a határvonal a két osztály közötti legnagyobb távolságot biztosítsa. Az algoritmus megtalálja az optimális hipersíket, amely a legnagyobb margin-t biztosítja, amely egyenlő a két legközelebbi adatpont távolságával.
- (3) Osztályozás: Az algoritmus az új adatokat a hipersíkon való pozíciójuk alapján osztályozza. Az algoritmus eldönti, hogy az új adat melyik oldalon található a határvonalon.

Az SVM előnye, hogy jól működik a magas dimenziós adatokon és olyan feladatokon, ahol az osztályok közötti határok nem lineárisak. Az SVM továbbá rendkívül hatékony az outlierek kezelésében, mivel csak azok a pontok határozzák meg a határvonalat, amelyek a legközelebb vannak hozzá. Az SVM azonban egy nehézkes algoritmus, amelynek tanítása hosszabb ideig tart, ha nagy adathalmazokat kell kezelni. Az algoritmus hiperparamétereinek finomhangolása továbbá kihívást jelenthet, különösen ha nem rendelkezünk megfelelő előzetes tudással az adathalmazról.

*SVM Paraméterek.* Több féle "kernel function" - szűrő - közül választhatunk, ami lehet lineáris, polinomiális, exponenciális stb. Mi az RBF (Radial Basis Function) exponenciális szűrőt használtuk, aminek két fő paramétere van:  $C$  és  $\gamma$ .  $C$  az SVM egy általános paramétere, ami másik szűrők használatakor is jelen van. Ez a paraméter határozza meg mennyire legyen elsimítva az osztályok közötti határ. Alacsony  $C$  sima határvonalhoz vezet, amíg a magas  $C$  az okozza, hogy minden tanító adatot pontosan osztályozni tudjon. A  $\gamma$  prarméter, azt határozza meg, hogy egy tanító adatnak makkora befolyása van. Magas  $\gamma$  esetén az egymáshoz közelebbi adatok magas befolyással bírnak egymásra. Mérésein során azt tapasztaltuk, hogy az SVM tényleg gyors és hatékony, még nagy tanító halmaz mellett is.

A *Decision Tree* (*döntési fa*). egy olyan algoritmus, amely a bemeneti adatok alapján egy hierarchikus fastruktúrát hoz létre. Az ilyen fastruktúra minden csomópontjában egy adatfajta tulajdonsága áll, és a levélcsoportokban pedig a végleges osztálycímek találhatóak. A döntési fában minden ág egy adott tulajdonság értékét reprezentálja, és a fa

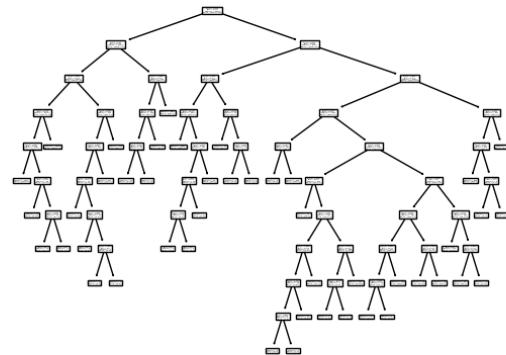


Fig. 14. Decision Tree Visualization

felépítése során az algoritmus igyekszik minél jobban felosztani a bemeneti adatokat az osztályok között. A Decision Tree osztályozó algoritmus létrehozása során a következő lépések szükségesek:

- (1) Adatok előkészítése: Az adatokat elő kell készíteni az osztályozó algoritmus számára, amely magában foglalhatja az adatok előfeldolgozását, a hiányzó értékek kezelését és a kategórikus változók átalakítását számszerű adatokká.
- (2) Fa építése: A faépítés során az algoritmus megpróbálja kiválasztani a legmegfelelőbb tulajdonságot a bemeneti adatok osztályozásához, majd a tulajdonság értékeitől függően felosztja az adatokat az algoritmus által meghatározott csoportokba. Az algoritmus folytatja ezt a folyamatot minden ágba, amíg el nem éri a megállási feltételt.
- (3) Fa értékelése: Az értékelés során az algoritmus az osztálycímeket rendel a bemeneti adatokhoz a fa segítségével.

A döntési fa osztályozó algoritmus előnye, hogy a modell könnyen értelmezhető és átlátható, így könnyen megérthető, hogy a modell hogyan dönt a különböző osztálycímekkel kapcsolatban. Az algoritmus továbbá jól alkalmazható kategórikus és numerikus változók kezelésére, valamint skálázható és gyorsan futtatható nagyobb adathalmazokon is. Egy döntési fát könnyű vizualizálni és kirajzolni, hisz egyszerű "if-else" logikai elágazásokból épül fel (lásd 14). Nagy és komplex fák alakulhatnak ki, amik túltanulást eredményezhetnek, és már kis eltérések a tanító halmazban nagy eltéréseket eredményezhetnek a fa logikájában.

## 5.5 Feature vektorok

Ahogyan klaszterezésnél is, fontos meghatározni egy olyan feature vektort, ami a legjobban jellemzi a trajektóriát. Ebben a papírban két fajta vektor verziót fogunk tárgyalni, amik

a tesztek során a legjobban teljesítettek. Az egyik a legelső verzió amit kipróbáltunk, a másik a hetedik verzió, ami jobban teljesített mint az összes többi 2-6 verzió.

*Az első verzió.* Felépítése a következő  $[x_0, y_0, v_{x_0}, v_{y_0}, x_m, y_m, x_l, y_l, v_{x_l}, v_{y_l}]$ , ahol  $m$  index jelöli az időben középen elhelyezkedő detektálást, az  $l$  index pedig az utolsó, legfrissebb detektálást jelöli. Ez a vektor jól reprezentálja a valós idejű futás közben keletkező trajektoriákat, mert egyszerre több száz detektálást objektumonként nem lehet eltárolni a memórában, hanem egy meghatározott méretű buffert kell alkalmazni, aminek mi 15 vagy 30 detektálást adtunk. Az adathalmazban eltárolt trajektoriák ennél a buffernél több detektálást tartalmaznak, ezért az első verzióról a trajektoriát  $k$  részre osztottuk, így egy trajektória szelet mérete  $s_n = n_d/k$ , ahol  $n_d$  a detektálások számossága a trajektoriában. Ezekből a szeletekből képeztük az egyes feature vektorokat.

*A hetedik verzió.* Felépítése  $[x_0 * w_1, y_0 * w_2, v_{x_0} * w_3, v_{y_0} * w_4, x_l * w_5, y_l * w_6, v_{x_l} * w_7, v_{y_l} * w_8]$ . Ennél a verzióról használtunk súlyokat, ahol  $w_1 = 1$ ,  $w_2 = 1$ ,  $w_3 = 100$ ,  $w_4 = 100$ ,  $w_5 = 2$ ,  $w_6 = 2$ ,  $w_7 = 200$ ,  $w_8 = 200$ . Mivel a sebességek két nagyságrenddel kisebbek mint a koordináták, ezért felszoroztuk őket 100-as súlyokkal. Hogy a 15-30 detektálás nagyságú bufferekben nagyobb hangsúlyt kapjanak a legfrissebb koordináták és sebességek, így azok 2-es és 200-as szorzót kaptak. A mérések eredményből azt lehet levonni, hogy a 30-as bufferméret használata nem kifizetődő, hiszen nem növekedett a pontosság, és kétszer akkora memória igénye van.

*Teszt eredmények.* Azt mutatják, hogy az utóbbi verzió növelte az SVM pontosságát, viszont rontott a KNN és DT pontosságát (lásd 3). Még negyobb adathalmazok esetén, ahol pár ezer trajektória helyett több tíz vagy akár százezer van, érdemes megfontolni, hogy a 7. verzióval tanítunk be SVM modellt, mivel a KNN futási ideje ekkora adatmennyiség esetén sokkal lassabb lesz.

**5.5.1 Adatdúsítás.** Hogy minél pontosabban reprezentáljuk a valód idejű futást és növeljük a tanító adathalmazt, egy trajektoriából több feature vektort állítunk elő. Ezeknek a számosságát, a feature vektorokat legeneráló algoritmusban szabtuk meg. Ezzel azt is szabályoztuk, hogy mekkora időszeletből prediktáljon a modellünk. Mint ahogyan nem is említettük, valós időben 15, max 30 detektálást érdemes tárolni a bufferben. A mérési eredmények azt mutatják, hogy 15 és a 30 nagyságú buffer között nincs nagy különbség pontosságban. Futási idő szempontjából érdemes lehet a 15 nagyságú buffert választani, ha van elég tanító adat, és időt akarunk spórolni tanításnál, akkor a 30 nagyságú buffert is választhatjuk, mivel így felére csökken a feature vektorok száma, ez kevesebb tanítási időt jelent, viszont futás közben lesz nagyobb a memóriaigény.

## 5.6 Pontosság mérése

A pontosság mérésére háromféle metrikát használtunk.

*Accuracy Score.* Ha  $\hat{y}_i$  az  $i$ . minta predikciója és  $y_i$  a hozzá tartozó valodi érték, akkor az eltalált predikciók és összes predikció hányszáma, amit így lehet leírni:

$$\text{accuracy}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} 1(\hat{y}_i = y_i) \quad (6)$$

*Balanced Accuracy.* amit ezért használtunk, hogy az adathalmaz kiegyensúlyozatlansága miatt ne kapjunk fals pontosságot. Ha minden osztályra egyenlően jól teljesít a klasszifikációs modellünk, akkor a sima Accuracy-t kapjuk vissza. Ha a teszt adathalmaz kiegyensúlyozatlansága miatt az egyik osztálynak jobb a pontossága mint egy másiknak, akkor ezt az értéket elosztja a számával. Ha az  $y_i$  a valodi értéke az  $i$ . mintának, és  $w_i$  a hozzá tartozó súly, akkor ezt a súlyt a következőképpen korrigáljuk:

$$\hat{w}_i = \frac{w_i}{\sum_j 1(y_j = y_i) w_j} \quad (7)$$

ahol  $1(x)$  a karakterisztikus függvény. Adott a  $\hat{y}_i$  predikció az  $i$ . mintának, így a balanced accuracy-t így definiálhatjuk:

$$\text{balanced-accuracy}(y, \hat{y}, w) = \frac{1}{\sum \hat{w}_i} \sum_i 1(\hat{y}_i = y_i) \hat{w}_i \quad (8)$$

*Top-K Accuracy.* az Accuracy Score egy generalizált változata. A különbség az, hogy a predikció akkor számít igaznak, ha beletartozik a  $k$  legmagasabb valószínűségű predíciók közé. Ha  $\hat{f}_{i,j}$  a  $i$ . mintának a  $j$ . legmagasabb predikciója, és  $y_i$  a hozzá tartozó valodi predikció, akkor az eltalált predikciók és az összes minta hányszáma így lehet definiálni:

$$\text{top-}k \text{ accuracy}(y, \hat{f}) = \frac{1}{n_{\text{samples}}} \sum_{i=0}^{n_{\text{samples}}-1} \sum_{j=1}^k 1(\hat{f}_{i,j} = y_i) \quad (9)$$

ahol  $k$  a megengedett találhatások száma, és  $1(x)$  a karakterisztikus függvény.

**5.6.1 Adathalmaz szétválasztás.** A pontosság méréséhez el kell választanunk egy teszt adathalmazt a tanító adathalmaztól, mivel ha azon az adathalmazon tesztelünk, amin tanítottunk akkor tökéletes pontosságot kapnánk eredményül. Ezt *Overfitting*-nek hívják. A szétválasztást egy általunk implementált algoritmus végzi, ami véletlen szám generátort használ a minták kiválasztásához. Az algoritmusnak meg lehet adni paraméterként a tanító adathalmaz méretét, és egy *seed* értéket, ami azért fontos, hogy meg lehessen ismételni a szétválasztást.

**5.6.2 Cross Validation.** A túltanítás elkerülése érdekében, alkalmaztunk egy elterjedt metódust a cross-validationot. A cross validáció egy olyan metódus, ahol a tanító adathalmazt  $k$  részre osztják, ebből a  $k$  részből egyet kiválasztanak validációra, így keletkezik egy tanító és validáló adathalmaz. A tanító adathalmazon betanítanak egy modellt, aminek a

pontosságát megmérík a validáló adathalmazon. Ezt az algoritmust  $k$ -szor ismétlik meg, úgy hogy minden rész egyszer legyen validáló adathalmaz. Ezeknek a méréseknek az átlag pontosságát szokták kiszámolni. A cross-validation ből származó eredmények a 4. és 5. táblázatban mutatjuk be.

**5.6.3 Teszthalmazos validáció.** Hogy meggyőződjünk arról, hogy biztosan nem tanítottuk túl a modelltünket, egy olyan teszt adathalmazon is le kell tesztelnünk, amit nem használtunk fel egyszer sem cross-validáció alatt tanításra. Ezzel a méressel bebizonyosodhatunk róla, hogy a modellünket nincsen bias. A teszthalmazos mérés eredményeit a 2. és 3. táblázatban mutatjuk be.

**5.6.4 Modellek tárolása.** A betanított modelleket joblib file-ként tároltuk el, amit később python-nal tudunk betölteni.

## 6 VALÓS IDEJŰ ALKALMAZÁS

A modellek pontosságának tesztelésére nem csak mérőszámokat alkalmaztunk, hanem egy vizualizációs alkalmazást is fejlesztettünk. Az alkalmazásnak meg kell adni a joblib modell fájlt és a videót amin tanítottuk. Ezeken kívül meg lehet adni mekkora detekció buffert használjon és, hogy a top mennyi predikciót rajzolja ki. A lejátszó kirajzolja a klaszterek kimeneti pontjait, és az autók középpontjával köti össze. A legvalószínűbb predikció zöldel a kevésbé valószínű pedig pirossal van kirajzolva (lásd 15 16 18 17). Az alkalmazás futását a mellékletben megadott videókon lehet megtekinteni.

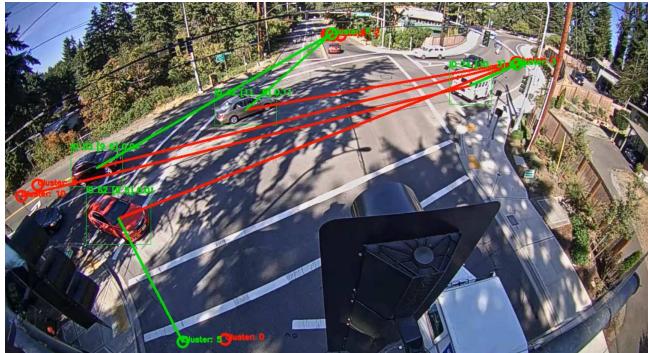


Fig. 15. Bellevue Newport real time application

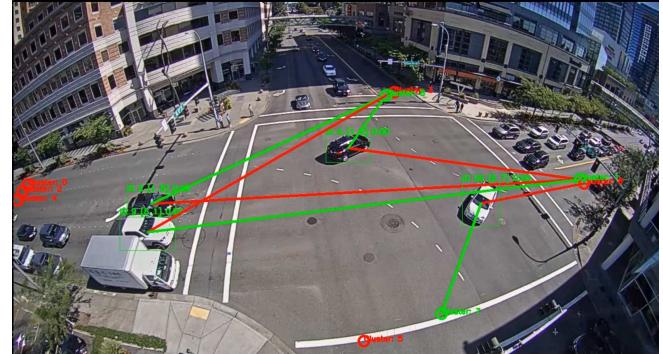


Fig. 16. Bellevue NE real time application

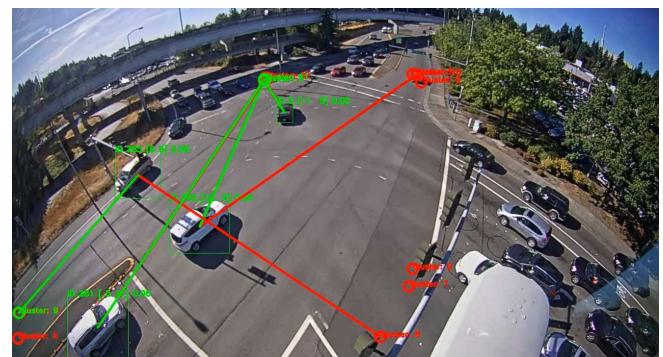


Fig. 17. Bellevue Eastgate real time application

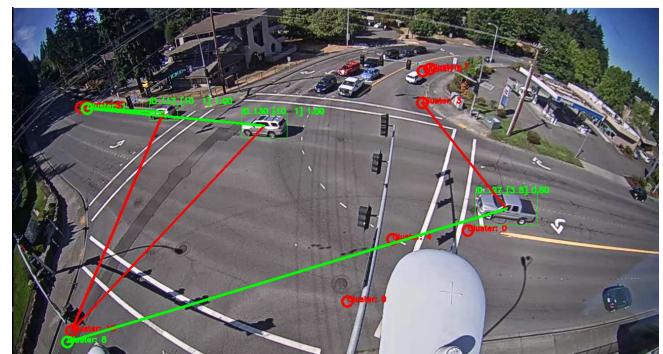


Fig. 18. Bellevue SE real time application

## 7 KONKLÚZIÓ

A kiépített keretrendszer a klaszterezéssel és klasszifikációs modellek betanításával, egy fontos lépés e terület fejlesztésében. A mérések jó alapot szolgálnak a jövőbeli kutatásoknak, milyen algoritmusokat érdemes még mélyebben megcvizsgálni, és hogy melyekkel nem érdemes a tobábiakban foglalkozni. Más objektumdetektáló és objektumkövető algoritmusokat is érdemes lehet kipróbálni, amivel az

adatgyűjtéstől fázist és az adattisztítási fázist lehet felgyorsítani és hatékonyabbá tenni. Az átlagos 90% pontosság amit elérünk a tanított modellekkel, még nem elég pontos, hogy biztonság kritikus rendszerekben alkalmazható legyen. Az adathalmaz növelésével ezt a jövőben növelni lehet. A feature vektorok dimenziószámának növelésével és másfajta szílyozással is növelhető ez a pontosság. A dimenzió növelése felveti a lehetőséget, hogy a jövőben mély neurális hálókat is teszteljünk az osztályozás feladatra. A megerősítő tanulás bevezetése a keretrendszerünké is egy nagy előrelépés lehet, ezt úgy lehetne megvalósítani, hogy valós idejű futás közben, a bejövő adatokon nem csak osztályozást végezünk hanem ezeket az adatokat folyamatosan mentjük, majd ha elég adat összegyűlt időközönként frissítjük a modellt, így a modell adaptálódni tud a forgalom változásához.

## REFERENCES

- D. Anguita, Luca Ghelardoni, Alessandro Ghio, L. Oneto, and Sandro Ridella. 2012. The 'K' in K-fold Cross Validation. In *The European Symposium on Artificial Neural Networks*.
- Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. *SIGMOD Rec.* 28, 2 (jun 1999), 49–60. <https://doi.org/10.1145/304181.304187>
- G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).
- L. Breiman, Jerome H. Friedman, Richard A. Olshen, and C. J. Stone. 1984. Classification and Regression Trees.
- Kay Henning Brodersen, Cheng Soon Ong, Klaas Enno Stephan, and Joachim M. Buhmann. 2010. The Balanced Accuracy and Its Posterior Distribution. In *Proceedings of the 2010 20th International Conference on Pattern Recognition (ICPR '10)*. IEEE Computer Society, USA, 3121–3124. <https://doi.org/10.1109/ICPR.2010.764>
- Tadeusz Caliński and Harabasz JA. 1974. A Dendrite Method for Cluster Analysis. *Communications in Statistics - Theory and Methods* 3 (01 1974), 1–27. <https://doi.org/10.1080/03610927408827101>
- Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology* 2 (2011), 27:1–27:27. Issue 3. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- David L. Davies and Donald W. Bouldin. 1979. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1, 2 (April 1979), 224–227. <https://doi.org/10.1109/TPAMI.1979.4766909>
- Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (Portland, Oregon) (KDD '96). AAAI Press, 226–231.
- Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. <https://doi.org/10.1038/s41586-020-2649-2>
- Richard D Hipp. 2020. SQLite. <https://www.sqlite.org/index.html>
- J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. <https://doi.org/10.1109/MCSE.2007.55>
- Joblib Development Team. 2020. *Joblib: running Python functions as pipeline jobs*. <https://joblib.readthedocs.io/>
- The pandas development team. 2020. *pandas-dev/pandas: Pandas*. <https://doi.org/10.5281/zenodo.3509134>
- Anand Paul, Naveen Chilamkurti, Alfred Daniel, and Seungmin Rho. 2017. Chapter 8 - Big Data collision analysis framework. In *Intelligent Vehicular Networks and Communications*, Anand Paul, Naveen Chilamkurti, Alfred Daniel, and Seungmin Rho (Eds.). Elsevier, 177–184. <https://doi.org/10.1016/B978-0-12-809266-8.00008-9>
- F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- Luca Rossi, Andrea Ajmar, Marina Paolanti, and Roberto Pierdicca. 2021. Vehicle trajectory prediction and generation using LSTM models and GANs. *PLOS ONE* 16, 7 (07 2021), 1–28. <https://doi.org/10.1371/journal.pone.0253868>
- Peter J. Rousseeuw. 1987. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.* 20 (1987), 53–65. [https://doi.org/10.1016/0377-0427\(87\)90125-7](https://doi.org/10.1016/0377-0427(87)90125-7)
- Erich Schubert, Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Trans. Database Syst.* 42, 3, Article 19 (jul 2017), 21 pages. <https://doi.org/10.1145/3068335>
- Chien-Yao Wang, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2022. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. *arXiv preprint arXiv:2207.02696* (2022).
- Nicolai Wojke and Alex Bewley. 2018. Deep Cosine Metric Learning for Person Re-identification. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE, 748–756. <https://doi.org/10.1109/WACV.2018.00087>
- Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (Montreal, Quebec, Canada) (SIGMOD '96). Association for Computing Machinery, New York, NY, USA, 103–114. <https://doi.org/10.1145/233269.233324>