

Széchenyi István Egyetem
Gépésmérnöki, Informatikai és Villamosmérnöki Kar

DIPLOAMUNKA

Péter Bence Gábor

Mérnök informatikus BSc szak

2023

A gerincen:

Péter Bence Gábor 2023 Nyilvános



Széchenyi István Egyetem
Gépészmérnöki, Informatikai
és Villamosmérnöki Kar

TDK Dolgozat

**Járművek trajektóriáinak előrejelzése
machine learning modellekkel**

Péter Bence Gábor

Konzulensek:

Dr. Horváth András egyetemi docens
Agg Áron PhD hallgató

Mérnök informatikus BSc szak

Győr, 2023

Kivonat

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 alkalmazásokban. Célunk, gépi látás és gépi tanulás felhasználásával, közlekedési csomópontok elemzésének automatizálása és felgyorsítása. A kutatás eredményeit, a kifejlesztett keretrendszerét és a felmerülő problémák megoldásait a gyakorlatban balesetek megelőzésére, renitens viselkedések kiszűrésére és forgalomirányító rendszerek támogatására lehet használni. A kutatásban egy trajektória osztályozó módszert ismertetünk, amely objektum detektálás és objektumkövetés segítségével elemezze a közlekedési csomópontokban elhaladó járművek mozgását. A mozgásuk alapján klaszterezí a trajektoriákat. A klaszterező algoritmus paramétereinek változtatásával különböző finomságú útvonal szétválasztás érhető el. Ezek az útvonal csoportok bemenetként szolgálnak az osztályozó algoritmus tanításánál, ami betanítás után valós időben tudja prediktálni a belépő járművek kimeneti pontjait. Az OPTICS klaszterező algoritmus bizonyult a mi esetünkben a leghatásosabbnak. Az OPTICS által csoportosított útvonalak végpontjai alapján további finomításra van lehetőség egy általunk létrehozott algoritmus segítségével, ami a KMeans algoritmust használja fel alapul. A közlekedők útvonalainak valós idejű előrejelzésére létrehozott módszerekben az SVM, KNN és DecisionTree osztályozó algoritmusok érték el a legmagasabb pontosságot az általunk végzett kiértékelések során. Továbbá az általunk kifejlesztett keretrendszer alkalmas forgalomszámlálásra és forgalmi statisztikák előállítására, ami értékes információként szolgálhat közlekedésmérnököknek.

Abstract

Intelligent transportation systems (ITS) are increasingly gaining importance nowadays, and they are being applied in various fields. Traffic node analysis is a popular topic in ITS applications. Our goal is to automate and accelerate traffic node analysis using machine vision and machine learning techniques. The research results, developed framework, and solutions to emerging problems can be used in practice to prevent accidents, detect aggressive driving, and support traffic management systems. We introduce a trajectory classification method that analyzes the movement of vehicles passing through traffic nodes using object detection and tracking. The algorithm clusters the trajectories based on their motion patterns. By adjusting the parameters of the clustering algorithm, we can achieve different levels of route separation. These routes serve as input to the classifier algorithm, which predicts the output points of incoming vehicles in real-time after training. The OPTICS clustering algorithm proved to be the most effective in our case. Using the endpoint of the routes clustered by OPTICS, we can further refine the classification with an algorithm we developed that uses the KMeans algorithm as a foundation. The methods we developed for real-time prediction of traffic routes achieved the highest accuracy in our evaluations, using the SVM, KNN, and Decision Tree classification algorithms. Additionally, our developed framework is suitable for traffic counting and generating traffic statistics, providing valuable information for transportation engineers.

Tartalomjegyzék

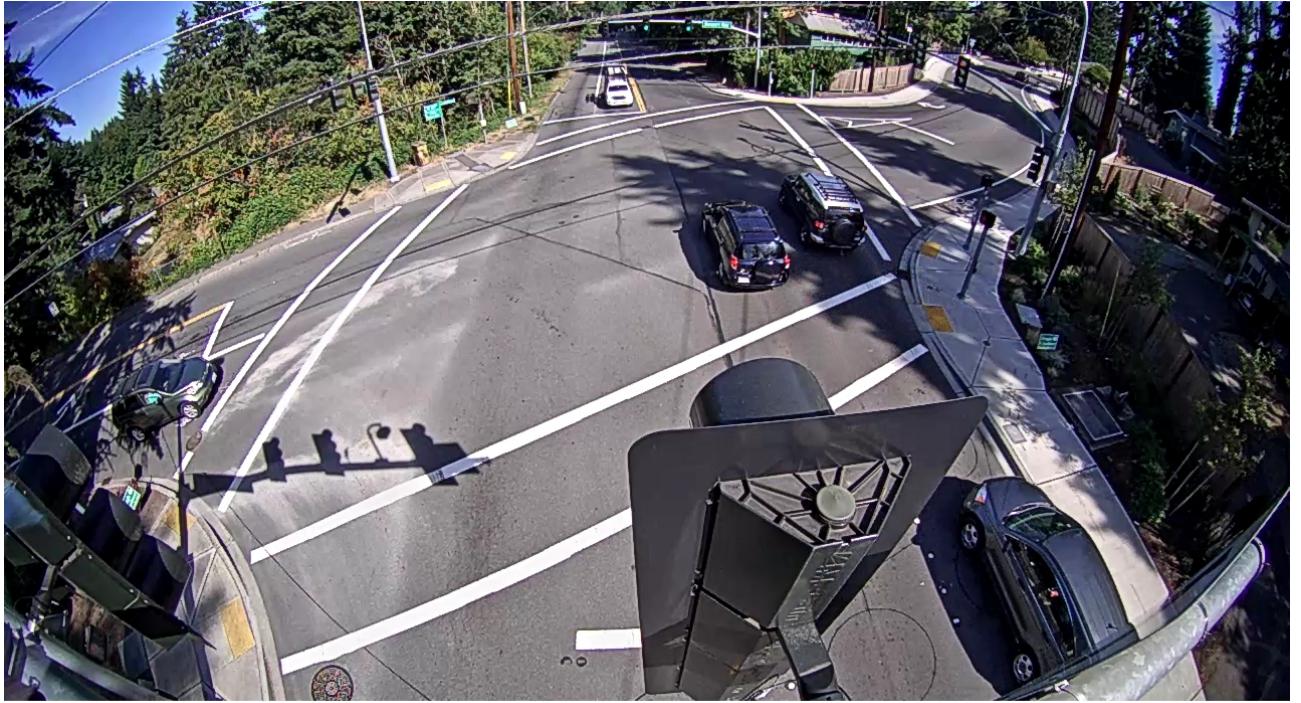
1. Bevezetés	4
2. Kapcsolódó kutatások	8
2.1. YOLO	8
2.2. DeepSORT	8
3. Adathalmaz kialakítása és feldolgozása	11
3.1. Adatstruktúra	12
3.2. Objektumdetektálás	15
3.2.1. YOLOv7	15
3.3. Objektumkövetés	15
3.3.1. DeepSORT	15
3.4. Objektum Orientált Implementáció	16
3.5. Implementáció dokumentálása	18
4. Klaszterezés	19
4.1. Adattisztítás	19
4.1.1. DeepSORT pontatlanság	21
4.1.2. Yolov7 paraméterek	21
4.2. Feature vektorok	25
4.3. Klaszterezési algoritmusok	25
4.4. Paraméterek kiválasztása	29
4.5. Közeli klaszterek egyesítése	30
5. Osztályozás	31
5.1. Multiclass	31
5.2. Binary	31
5.3. OneVsRest	32
5.4. Machine Learning modellek	32
5.5. Feature vektorok	35
5.5.1. Adatdúsítás	36
5.6. Pontosság mérése	36
5.6.1. Adathalmaz szétválasztás	37
5.6.2. Cross Validation - Kereszt validáció	37
5.6.3. Teszthalmazos validáció	37
5.6.4. Modellek tárolása	38
6. Valós idejű alkalmazás	39
7. Forgalmi statisztikák	41
7.1. Hisztogram	41
7.2. Hőtérkép	43
8. Konklúzió	45

1. Bevezetés

A nagy számú jármű, útvonal és csomópont miatt az is nagy kihívás, hogy számszerű és pontos képet kapjunk a közlekedés dinamikájáról, az egyes útvonalak leterheltségéről és a közleketők útvonal-választási szokásairól. Ennek követésére számtalan tradicionális módszer is létezik (pl. ember által végzett forgalom-számlálás). Dolgozatomban egy olyan módszert mutatunk be, mely erre a kérdésre a már fent leírt forgalomfigyelő kamerák képének elemzésével pontos, részletes eredményeket ad.

Az ITS (intelligent transportation system) fejlesztése a városokban erre megoldást jelenthet. Ez magában foglalja az információs és kommunikációs technológiák, mint például szenzorok, kamerák, kommunikációs hálózatok és adatelemzés 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 nagy kihívás és egyre növekszik irántuk a kereslet. E kutatási terület kiforraltanságából 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 törekünk egy módszertant és keretrendszert 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 termékajánlásokban, 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, úgynevezett klaszterezéssel határozzuk meg. Erre a feladatra KMeans, BIRCH [24] és DBSCAN [11][20], OPTICS [3] 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* [7] és *Top-k Accuracy Score*. Mindegyik mérőszám kiszámolásához *K-Fold Cross-Validation* [2] metódust alkalmaztunk, ahol $K = 5$.

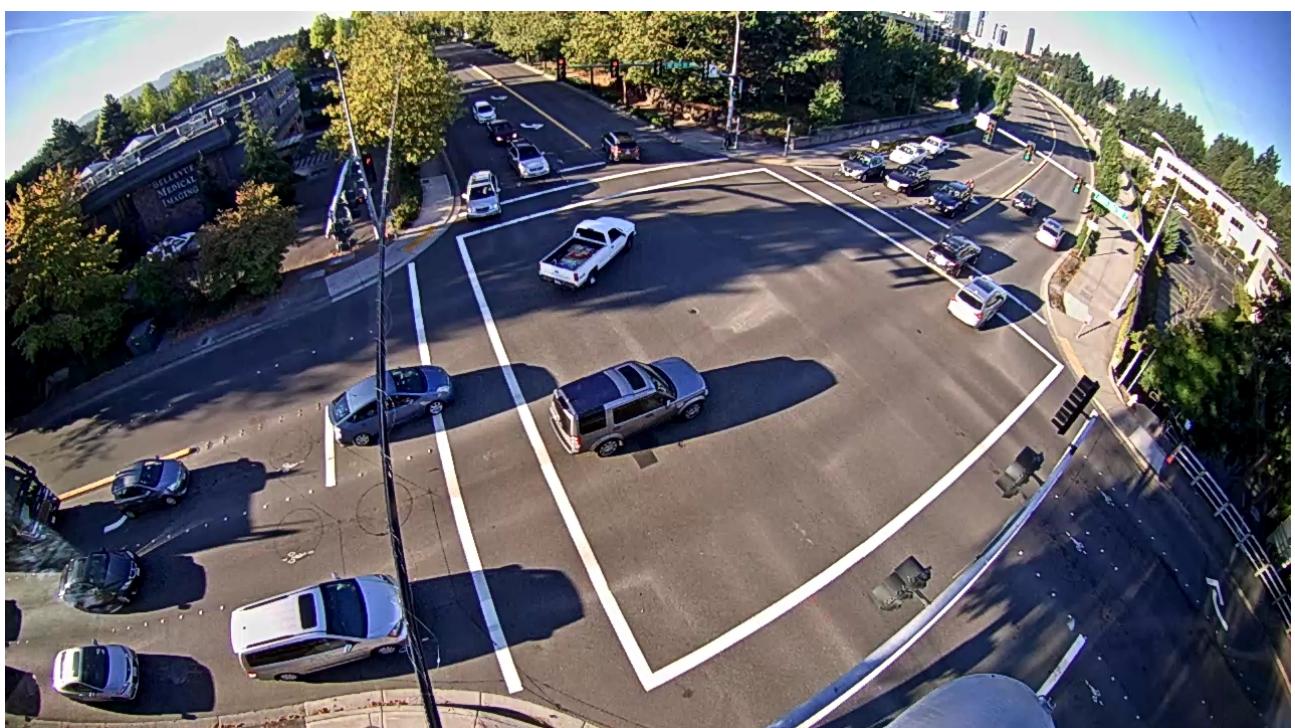


1. ábra. Bellevue Newport kereszteződés

A tanító adatok előállításához, objektumok detektálására a YOLOv7 [22] konvolúciós neurális hálót használtuk, ez a konvolúciós 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 [23] nevezetű algoritmust használtuk, ez kálmán filtert és konvolúciós 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 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 lásd 5. A videók pontos elérhetőségét a mellékletben található *urls.txt*-ben adtuk meg.



2. ábra. Bellevue Eastgate kereszteződés



3. ábra. Bellevue NE kereszteződés



Mon Sep 11 2017 08:14:19

4. ábra. Bellevue SE kereszteződés



5. ábra. La Grange KY North

2. Kapcsolódó kutatások

Sok ITS-el kapcsolatos kutatásban tárgyalják a forgalom folyás (traffic flow) előrejelzését. [16] összehasonlítja az eddig kutatott és használt modellek, mint például Kálmán-szűrő, 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 ez a terület belépett a *Big Data* korszakába. [18] 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. A 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: A 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: A YOLO algoritmus egy kiterjedt konvolúciós hálózatot alkalmaz a vektorra, amelynek célja az objektumok lokalizálása és azok osztályozása.
4. Objektum lokalizálása és osztályozása: A YOLO algoritmus az általa előállított tenzoron keresztül végzi az objektumok lokalizálását és azok osztályozását. Az algoritmus meghatározza az objektumok koordinátáit és a hozzájuk tartozó osztályt.

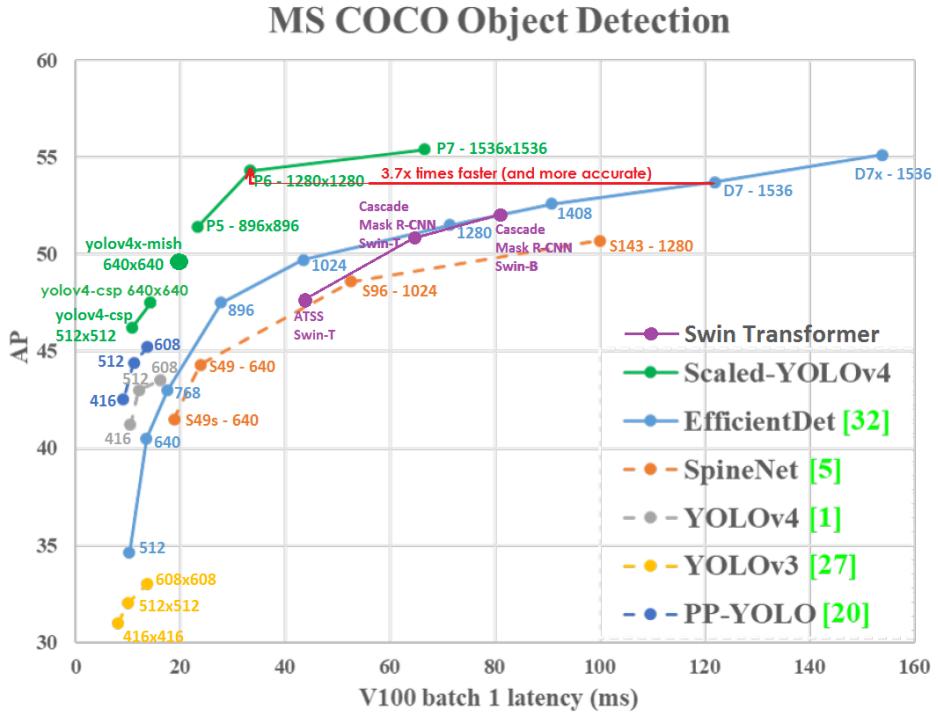
A YOLO algoritmus előnye, hogy nagyon gyors és hatékonyan kezeli az objektumok lokalizálását és azok osztályozását. Az algoritmus gyakran jobb teljesítményt nyújt, mint a hasonló módszerek, és a különböző objektumokat a bemeneti képen gyorsan és hatékonyan azonosítja. Azonban az YOLO algoritmus hibázhat, 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 sebességekben és pontosságban a modern konvolúciós hálókat (lásd 6). Beágyazott rendszerekben és videókártyákon is egyaránt jó a teljesítménye, ezért az ITS területén alkalmazható.

2.2. DeepSORT

A DeepSORT algoritmus működése a következő lépésekkel áll:

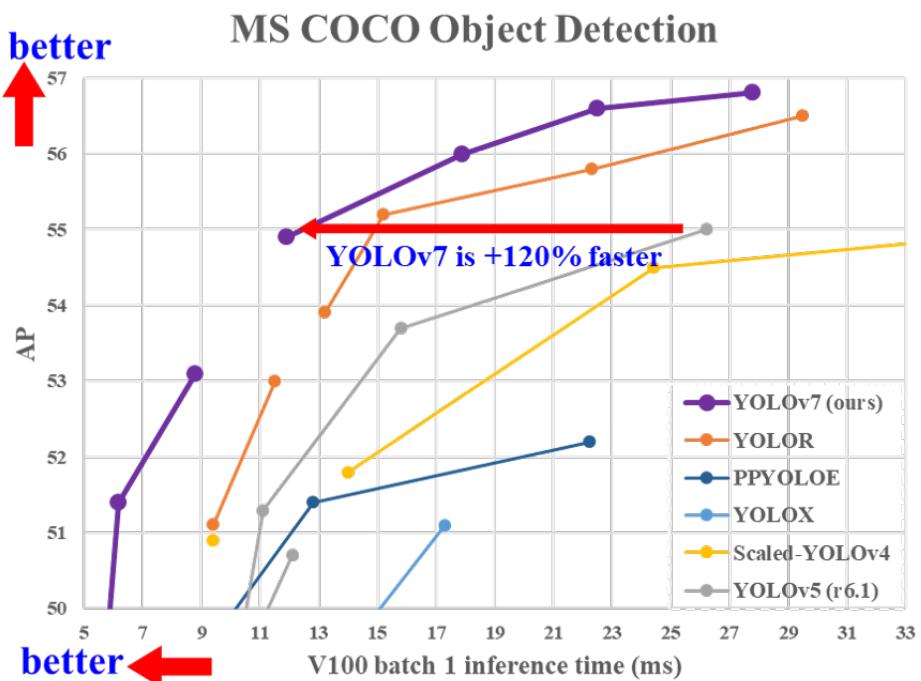
1. Objektumdetektálás: Az algoritmus először objektumdetektálással azonosítja az összes objektumot a videófelvételen, például a YOLO objektumdetektáló algoritmust használva.
2. Jellemzők kinyerése: A DeepSORT az objektumok jellemzőit (pl. méret, sebesség, szín) kinyeri, hogy a következő lépésben a következő objektumot azonosítani tudja.
3. Objektumazonosítás: Az algoritmus használ egy "tracklet" nevű algoritmust, hogy azonosítsa és kövesse az objektumokat az időben. A "tracklet" az objektum jellemzőit használja, hogy azonosítsa az adott objektumot a videófelvétel további részein.

YOLOv4 performance over other CNN models.



6. ábra. YOLOv4 Performance

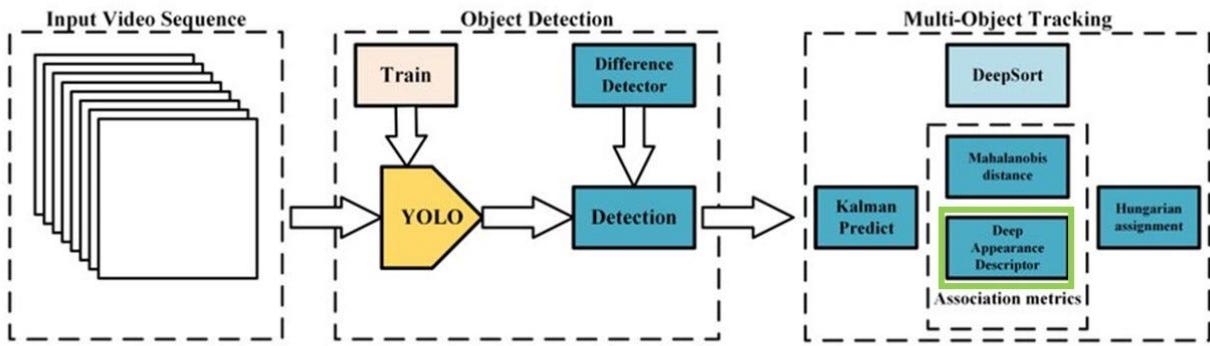
YOLOv7 performance over other yolo models.



7. ábra. YOLOv7 Performance

DeepSORT

Where is the Deep Learning
in all of this?



8. ábra. Objektum követés

4. Címkézés: Az objektumokat azonosítják egyedi azonosítókkal, hogy az algoritmus megkülönböztethesse azokat az egyes videófelvételeken.
5. Korszakosítás: A DeepSORT algoritmus általában a Kálmán-szűrőt használja, amely folyamatosan frissíti az objektumok helyzetének becslését. A Kálmán-szűrő segít az algoritmusnak megjósolni az objektumok további helyzetét a videófelvétel 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. Adathalmaz kialakítása és feldolgozása

A kutatás során saját adathalmazok kialakítására volt szükség. Az adatok begyűjtésére és eltárolására saját alkalmazást és keretrendszer fejlesztettünk ki. A szoftver keretrendszeret python nyelven írtuk meg, a forráskód ezen a linken megtalálható http://github.com/Pecneb/computer_vision_research. A fejlesztés során a következő programkönyvtárakat használtuk OpenCV [4], Numpy [12], Pandas [21], Scikit-Learn [17], Matplotlib [14], SQLite [13], Joblib [15]. Az adathalmazokat SQLite adatbázisban és joblib fájlokban tároltuk el.

OpenCV A videók feldolgozásához és a képkockák feldolgozásához használtuk az OpenCV könyvtárat. Az OpenCV egy nyílt forráskódú programkönyvtár, amelyet a gépi látás és a gépi tanulás alkalmazásokhoz fejlesztettek ki. Az OpenCV-t C++ nyelven írták, de támogatja a Python, Java és MATLAB programozási nyelveket is. Az OpenCV-t a BSD licenc alatt terjesztik, és szabadon használható és módosítható.

Numpy A Numpy egy Python könyvtár, amelyet a tudományos számításokhoz fejlesztettek ki. A Numpy-t C és Fortran nyelveken írták. A Numpy-t a tudományos számításokhoz, például a mátrixműveletekhez, a mátrixok létrehozásához és a lineáris algebrai műveletekhez használják.

Pandas A Pandas egy Python könyvtár, amelyet a nagy teljesítményű adatelemzéshez és adatmanipulációhoz fejlesztettek ki. Numpy könyvtárra épül.

Scikit-Learn A Scikit-Learn egy Python könyvtár, amelyet a gépi tanuláshoz fejlesztettek ki. A Scikit-Learn a Numpy és a SciPy könyvtárakra épül.

Matplotlib A Matplotlib egy Python könyvtár, amelyet a képek és grafikonok megjelenítéséhez fejlesztettek ki. A Matplotlib-et a Numpy könyvtárra építették.

SQLite Az SQLite egy nyílt forráskódú relációs adatbázis-kezelő rendszer, amelyet a beágyazott adatbázisokhoz fejlesztettek ki. Az SQLite-t C nyelven írták.

Joblib A Joblib egy Python könyvtár, amelyet a Python objektumok hatékony szerializálásához és deserializálásához, számításigényes feladatok paralellizálására fejlesztettek ki. A Joblib-et a Numpy könyvtárra építették.

3.1. Adatstruktúra

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

SQL schema Az SQL schema a következőképpen néz ki:

```
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 lettek kiszervezve, ahol az *objID* idegen kulccsal 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* koordinátáját, az objektum szélességét *width* és magasságát *width*, sebességét *v_x, v_y* és gyorsulását *a_x, a_y*, amik a deepSORT által kalkulált értékek, így még külön a koordinátákból kiszámolt *v_{x_c}*, v_{y_c sebességet és *a_{x_c}*, a_{y_c 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 videókép szélesség *w*, magasság *h*, akkor a képarány *r = w/h*, és az eltárolt koordináták}}

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

$$v_x = \frac{v_{x_0}}{w} \cdot r, v_y = \frac{v_{y_0}}{w} \cdot r \quad (2)$$

$$a_x = \frac{a_{x_0}}{w} \cdot r, a_y = \frac{a_{y_0}}{w} \cdot r \quad (3)$$

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

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

Python osztályok A python osztályokhoz a python beépített Dataclass könyvtárat használtam, ami sok boilerplate kódot spórol meg. Detektálások és Trajektóriák reprezentálására, összehasonlítására és numpy-vel való műveletekre nagyon hasznos.

Python Detection osztály

```

@dataclass
class Detection:
    label: str
    confidence: float
    X: float
    Y: float
    Width: float
    Height: float
    frameID: int
    VX: float = field(init=False)
    VY: float = field(init=False)
    AX: float = field(init=False)
    AY: float = field(init=False)
    objID: int = field(init=False)

    def __repr__(self) -> str:
        return f"Label: {self.label}, \
            Confidence: {self.confidence}, \
            X: {self.X}, Y: {self.Y}, Width: \
            {self.Width}, Height: {self.Height}, Framenumber: {self.frameID}"

    def __eq__(self, other) -> bool:
        if self.label != other.label:
            return False
        if self.confidence != other.confidence:
            return False
        if self.X != other.X:
            return False
        if self.Y != other.Y:
            return False
        if self.Width != other.Width:
            return False

```

```

        if self.Height != other.Height:
            return False
        if self.frameID != other.frameID:
            return False
        return True
    
```

Python TrackedObject osztály tárolja el egy objektum összes detektálását, és a detektálásokból kiszámolt sebességeket és gyorsulásokat. A trajektória osztály számon tartja az objektum állapotát, hogy mennyi ideje történt az utolsó detektálás és hogy mozog-e vagy sem.

```

@dataclass
class TrackedObject:
    objID: int
    label: int = field(init=False)
    futureX: list = field(init=False)
    futureY: list = field(init=False)
    history: List[Detection] = field(init=False)
    history_X: np.ndarray = field(init=False)
    history_Y: np.ndarray = field(init=False)
    history_VX_calculated: np.ndarray = field(init=False)
    history_VY_calculated: np.ndarray = field(init=False)
    history_AX_calculated: np.ndarray = field(init=False)
    history_AY_calculated: np.ndarray = field(init=False)
    isMoving: bool = field(init=False)
    time_since_update: int = field(init=False)
    max_age: int
    mean: list = field(init=False)
    X: int
    Y: int
    VX: float = field(init=False)
    VY: float = field(init=False)
    AX: float = field(init=False)
    AY: float = field(init=False)
    _dataset: str = field(init=False)

    def __init__(self, id: int, first: Detection, max_age: int = 30):
        self.objID = id
        self.history = [first]
        self.history_X = np.array([first.X])
        self.history_Y = np.array([first.Y])
        self.history_VX_calculated = np.array([])
        self.history_VY_calculated = np.array([])
        self.history_VT = np.array([])
        self.history_AX_calculated = np.array([])
        self.history_AY_calculated = np.array([])
        self.X = first.X
        self.Y = first.Y
        self.VX = 0
        self.VY = 0
        self.AX = 0
    
```

```

    self.AY = 0
    self.history[-1].VX = self.VX
    self.history[-1].VY = self.VY
    self.history[-1].AX = self.AX
    self.history[-1].AY = self.AY
    self.label = first.label
    self.isMoving = False
    self.max_age = max_age
    self.time_since_update = 0
    self.mean = []
    self._dataset = ""

```

3.2. Objektumdetektálás

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

3.2.1. YOLOv7

A Yolov7 pythonban azon belül is pytorch-ban van implementálva. Hogy használni tudjam saját API-t kellett fejlesztenem hozzá. Az API része a képek előfeldolgozása, beadása a neurális hálózatba majd utófeldolgozása, hogy emberileg értelmezhető detektálásokat kapjunk. (lásd 7). Yolov7 felhasználásához objektum orientált megközelítést használtam. Egy Yolov7 osztályt hoztam létre, hogy elvégezze a detektálási feleadaokat, és könnyen olvasható és használható legyen.

3.3. Objektumkövetés

Ahhoz, hogy trajektóriák alapján tudjunk szabályosságokat 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. Yolo-hoz hasonlóan egy oszályként implementáltam az objektumkövetési interface-t.

3.4. Objektum Orientált Implementáció

Az objektum detektálási és követése pipeline-t egy objektum orientált megközelítéssel valósítottam meg. A programban 4 fő osztályt hoztam létre. A osztály architektúra diagramja 9 ábrán látható. A részletes implementációs dokumentáció a mellékletben található. A forráskód pedig a korábban említett github oldalon található http://github.com/Pecneb/computer-vision_research.

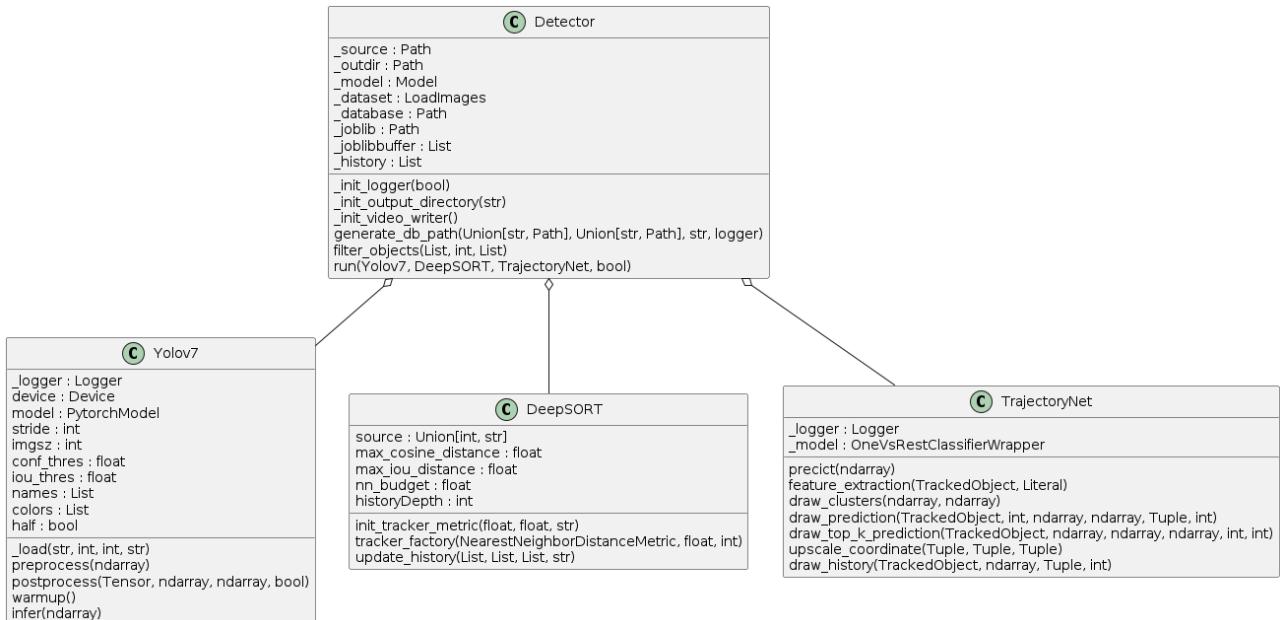
Yolov7 osztály felelős a képek előfeldolgozásáért, az objektumok detektálásáról, majd utófeldolgozásáról.

DeepSORT osztály felelős a detektálások követéséért, és a trajektóriák kialakításáról.

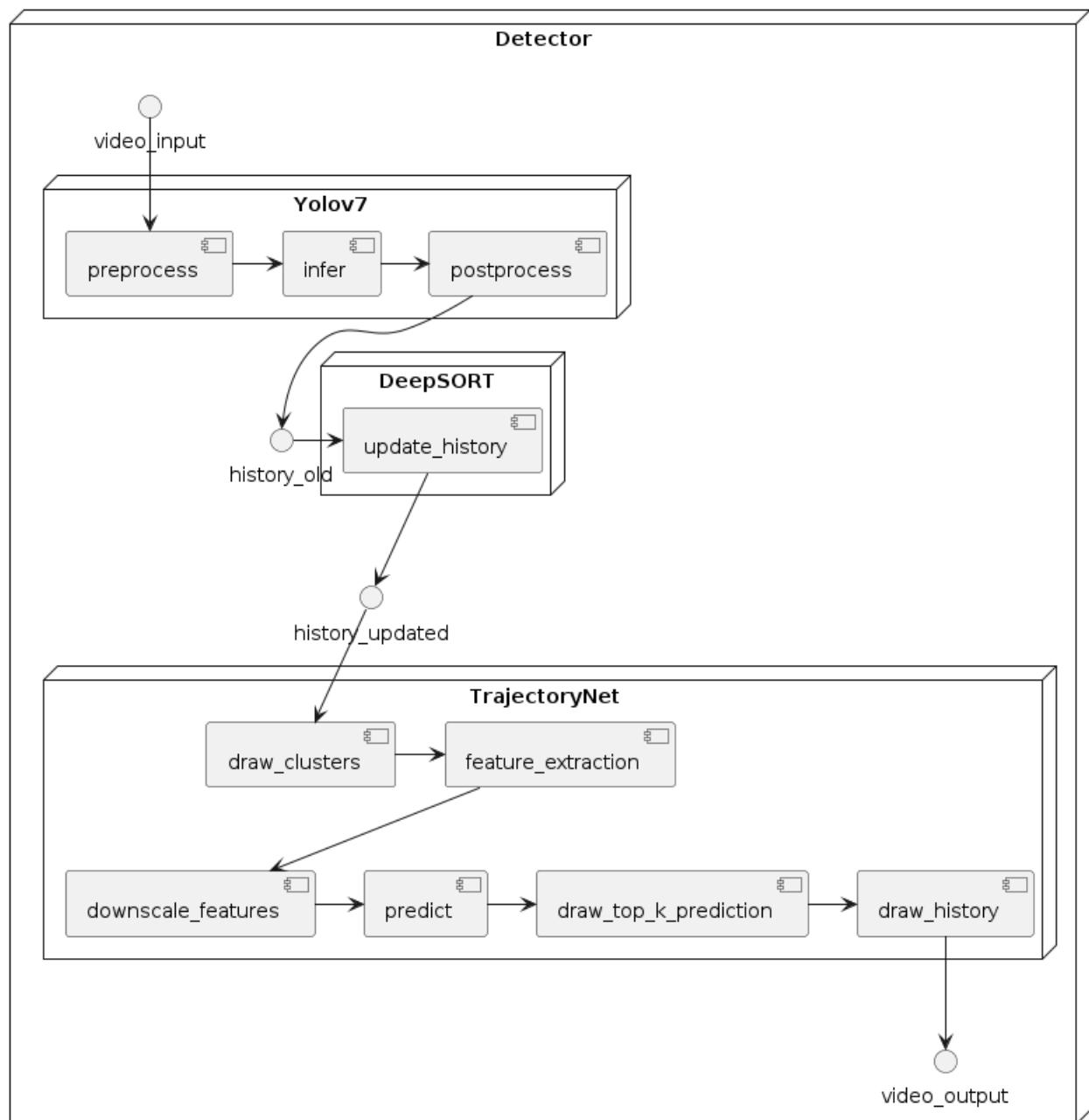
TrajectoryNet osztály kezeli az általunk tanított gépi tanulási modelleket, a feature vektorok előállítását és osztályozását.

Detector osztály fogja össze az előbb felsorolt osztályokat. Inicializálja a bemeneti és kimeneti adatokat. Végig iterál a videó képkockáin és futtatja a detektáló és követő algoritmusokat.

Detector *run()* metódusban implementáltam az objektum detektálás, követés és trajektória predikciós pipeline-t (lásd 10). A kép egy képkocka feldolgozását mutatja be.



9. ábra. Osztály diagram



10. ábra. Komponens diagram

3.5. Implementáció dokumentálása

A programok dokumentálására a Sphinx [5] keretrendszer használtuk. A Sphinx egy dokumentációs generátor, amely a megfelelően dokumentált Python kódból (ezalatt a Python nyelvhez tartozó dokumentációs sztringeket értjük) HTML és Latex dokumentációt készít. Egy ilyen dokumentációs sztringre példa.

```
def make_4D_feature_vectors(trackedObjects: List) -> np.ndarray:
    """
    Create 4D feature vectors from tracks.

    Parameters
    -----
    trackedObjects : list
        List of tracked objects.

    Returns
    -----
    np.ndarray
        Numpy array of feature vectors.

    Notes
    -----
    The enter and exit coordinates are put in one vector, creating 4D vectors.
    v = [enterX, enterY, exitX, exitY]
    """
    featureVectors = np.array([np.array(
        [obj.history[0].X, obj.history[0].Y, obj.history[-1].X, obj.history[-1].Y])
        for obj in tqdm.tqdm(trackedObjects, desc="Feature vectors.")])
    return featureVectors
```

4. Klaszterezés

A klaszterezés segítségével lehet az adathalmazból előállítani az osztályozás alapjául szolgáló csoportokat. 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 közel, 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 általam implementált szűrő algoritmust használtam az adathalmazon.

Trajektória hossz Elsőnek a trajektóriák belépő és kilépő pontjainak az euklideszi távolsága alapján szűrtünk. Ezzel a szűréssel a zajos, félbeszakadt trajektóriákat szűrjük ki.

```
def euclidean_distance(q1: float, p1: float, q2: float, p2: float):
    return (((q1-p1)**2)+((q2-p2)**2))**0.5

def filter_out_false_positive_detections_by_enter_exit_distance(
    trackedObjects, threshold):
    filteredTracks = list(filterfalse(lambda obj: euclidean_distance(
        obj.history[0].X, obj.history[-1].X, obj.history[0].Y, obj.history[-1].Y)
        < threshold, trackedObjects))
    return filteredTracks
```

Itt az euklideszi távolságot saját magam implementáltam, mert ez a megoldás bizonyult a leggyorsabbnak egyszerű, két 2 dimenziós pont között. A numpy-nak több féle megoldását és a scipy könyvtárban található távolságszámítást is kipróbáltam, de ezek lassabbnak bizonyultak. A méréshez a *timeit* könyvtárat használtam. A mérés eredményeit a 11 ábrán láthatjuk. A méréshez minden módszert 1000000 alkalommal futtattam le, és az átlagot vettettem. Az *own* címke jelenti a saját implementáció áltagos futás idejét. A forrás kód a következő a numpy és scipy könyvtárakat használó függvényekhez:

```

def euclidean_distance(a, b):
    return (((a[0]-b[0])**2)+((a[1]-b[1])**2))**0.5

def euclidean_distance_np_linalg(a, b):
    return np.linalg.norm(a-b)

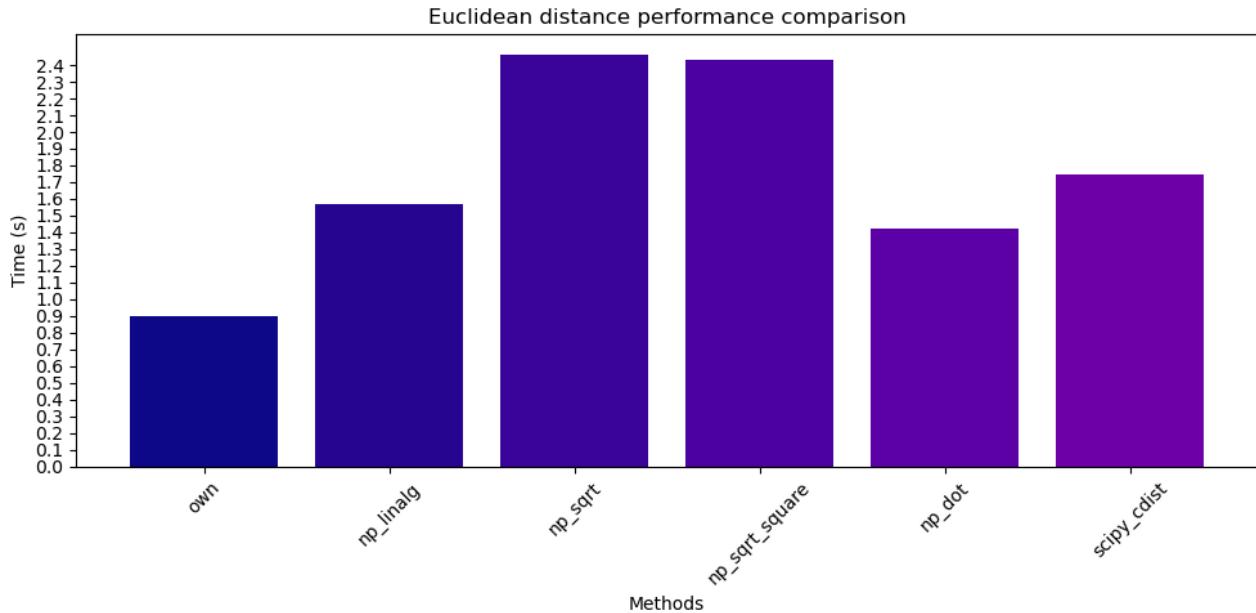
def euclidean_distance_np_sqrt(a, b):
    return np.sqrt(np.sum((a-b)**2))

def euclidean_distance_np_sqrt_square(a, b):
    return np.sqrt(np.sum(np.square(a-b)))

def euclidean_distance_np_dot(a, b):
    return np.sqrt((a-b).T @ (a-b))

def euclidean_distance_cdist(a, b):
    return cdist(a, b, metric='euclidean')

```



11. ábra. Euklideszi távolság sebesség mérés

Relatív szélekhez képesti távolság Majd a kép széleit meghatározzuk min-max kiválasztással, és azokat a trajektóriákat választjuk ki, amiknek a szélektől meghatározott távolságra vannak a belépő és kilépő pontjaik. Erre azért van szükség, mert ha belegondolunk, lehet hogy a kép egyik szélén valami eltakarja az utat pl. egy ház, és akkor az autók belépési pontjai a kép közepétől indulnak, ami nem jelenti azt, hogy az egy zajos, hibás trajektória, hanem ez a kereszteződés egy sajátossága.

```

def search_min_max_coordinates(trackedObjects):
    X = np.concatenate([o.history_X for o in trackedObjects], axis=None)
    Y = np.concatenate([o.history_Y for o in trackedObjects], axis=None)
    min_x = np.min(X)
    max_x = np.max(X)
    min_y = np.min(Y)
    max_y = np.max(Y)
    return min_x, min_y, max_x, max_y

def filter_out_edge_detections(trackedObjects, threshold):
    min_x, min_y, max_x, max_y = search_min_max_coordinates(trackedObjects)
    filteredTracks = []
    for obj in tqdm.tqdm(trackedObjects, desc="Filter out edge detections."):
        if (((obj.history[0].X <= min_x+threshold or
            obj.history[0].X >= max_x-threshold) or
            (obj.history[0].Y <= min_y+threshold or
            obj.history[0].Y >= max_y-threshold)) and
            ((obj.history[-1].X <= min_x+threshold or
            obj.history[-1].X >= max_x-threshold) or
            (obj.history[-1].Y <= min_y+threshold or
            obj.history[-1].Y >= max_y-threshold))):
            filteredTracks.append(obj)
    return filteredTracks

```

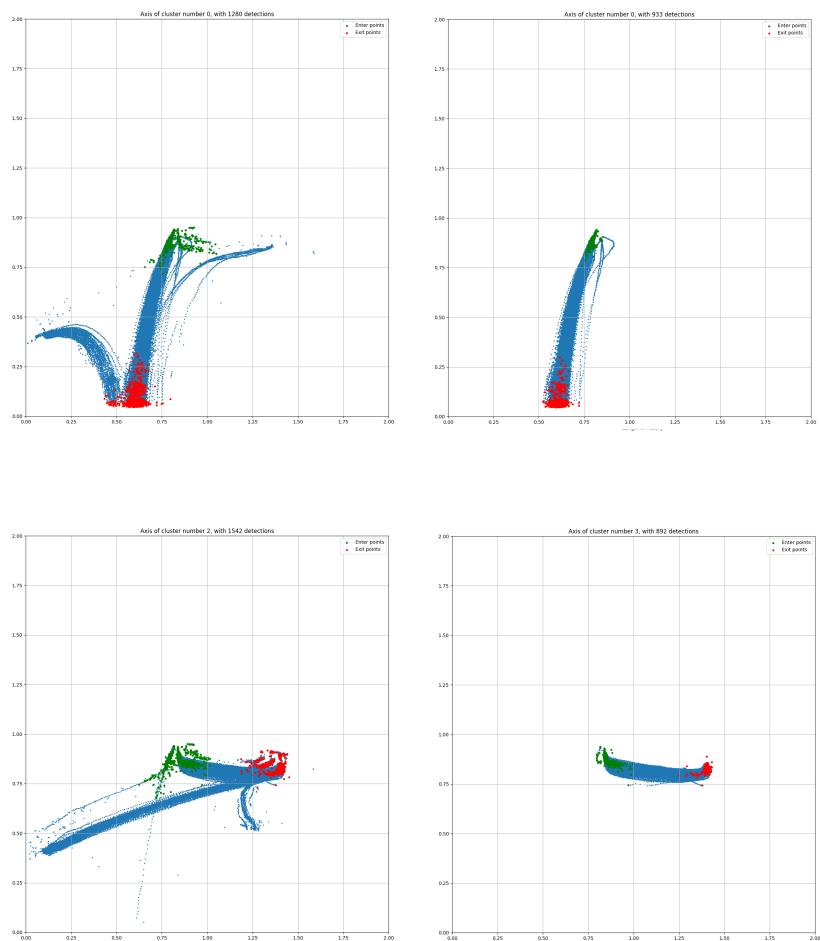
4.1.1. DeepSORT pontatlanság

Kutatásunk során azt tapasztaltuk, hogy a DeepSORT és a YOLO pontatlanságai felerősítik egymást. A YOLO hajlamos néha táblákat vagy rendőrlámpákat 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 utólagos szűréssel kellett korrigálnunk ezt a hibát.

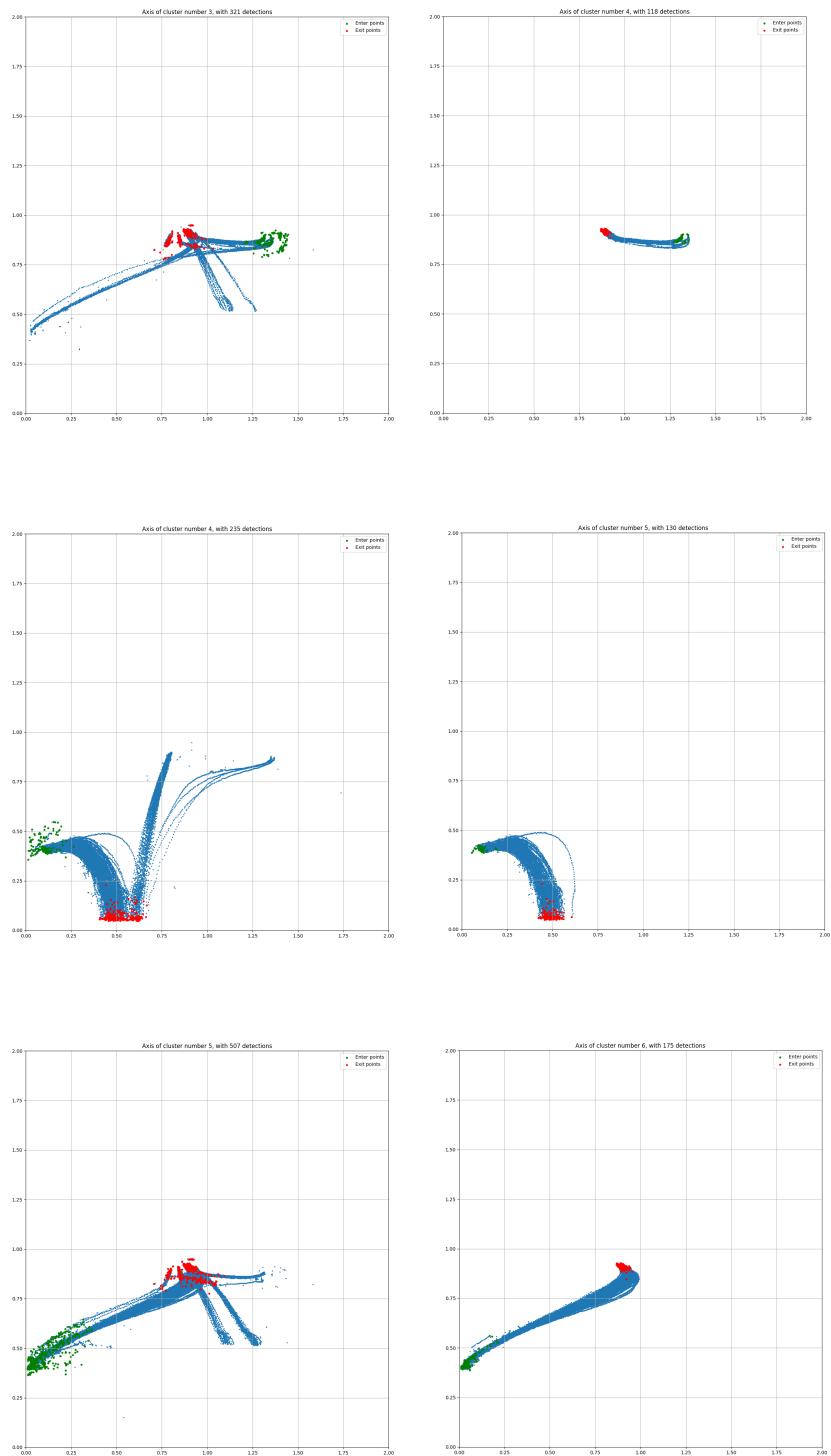
Áttapadásos zaj szűrése Az algoritmus végig iterál a trajektóriák pontjain és kiszámítja az egymást követő detektálások euklideszi távoláságát, és ha egy küszöbérték felett vannak, akkor eldobjuk a trajektóriát. Ez azért szükséges, mert a DeepSORT képes egymás közeli objektumokat összekeverni, és egyik objektumról a másikra áttapadni, ami hibás trajektóriákat eredményez. A következő képeken láthatók a klaszterek szűrés előtt és után (lásd 12 13 14), ahol a bal oldali oszlop reprezentálja a szűrés előtti klasztereket, a jobb oldali oszlop pedig a szűrés utáni klasztereket.

4.1.2. Yolov7 paraméterek

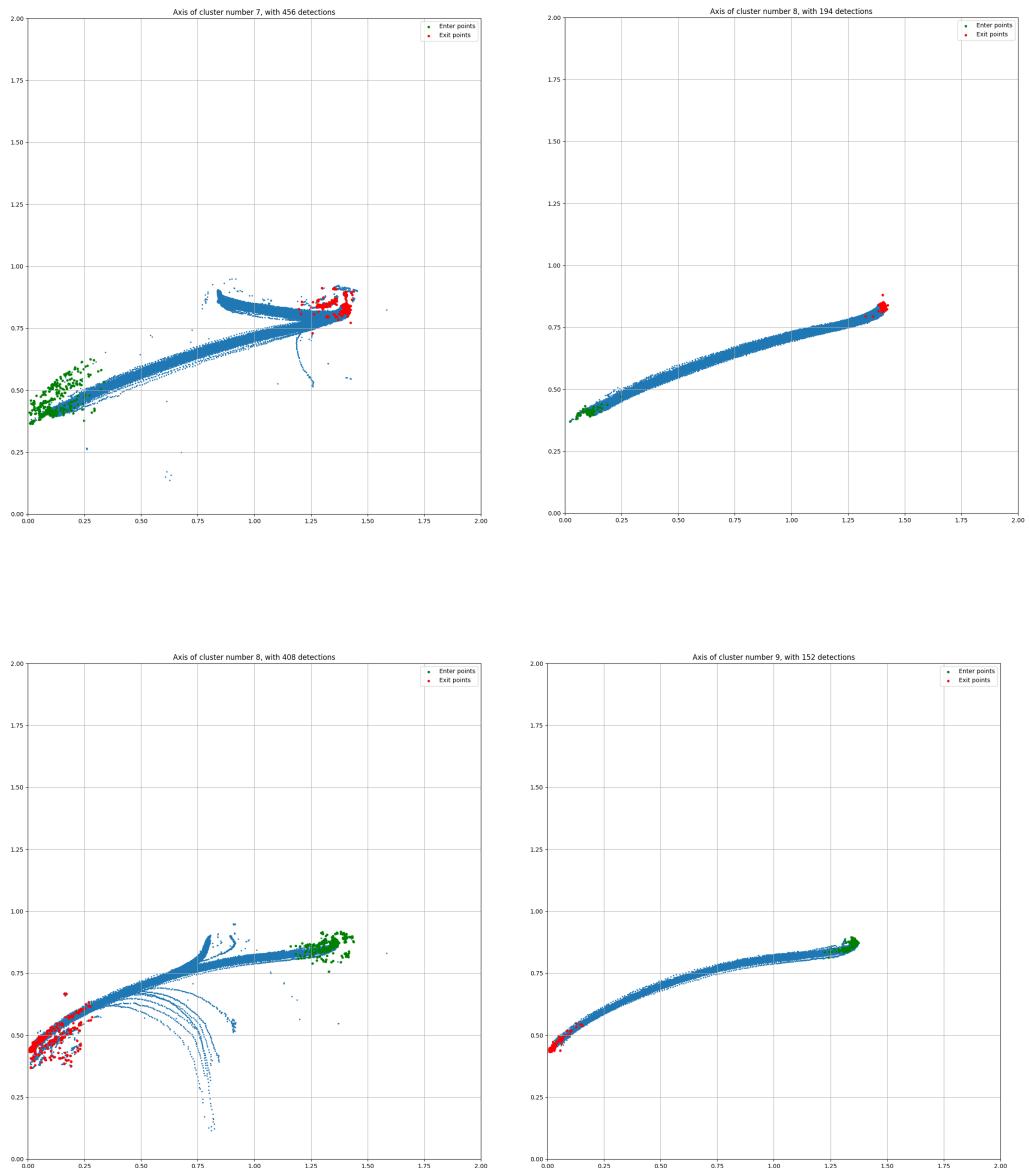
A Yolo-nak főbb paraméterei a konfidencia és az IoU (Intersection over Union). A konfidencia azt jelenti, hogy a neurális hálózat mennyire biztos abban, hogy az adott objektumot azonosította. Az IoU pedig azt jelenti, hogy a neurális hálózat mennyire biztos az objektum helyzetében. Ezen paraméterek finomhangolásával is javítani lehet a zajos detektálásokon és trajektóriákon.



12. ábra. Klaszterezés szűrő (piros: kilépő pont, zöld: belépő pont)



13. ábra. Klaszterezés szűrő



14. ábra. Klaszterezés szűrő

4.2. Feature vektorok

Klaszterezéshez 4 és 6 dimenziós feature vektorokat használ-tunk. 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 hatékonyab-bnak 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].

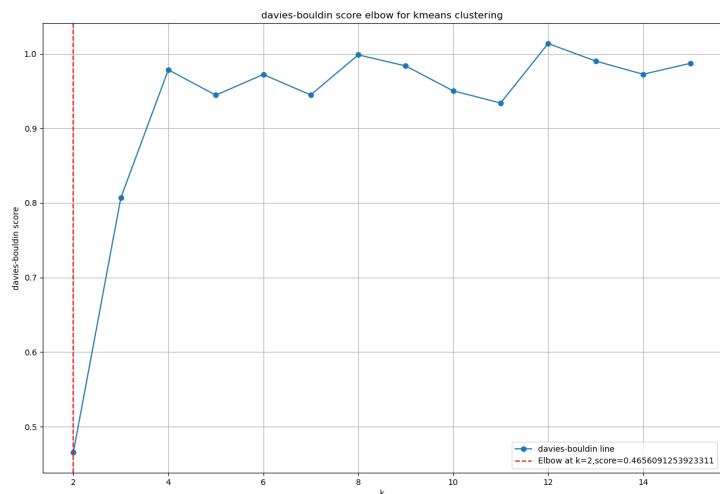
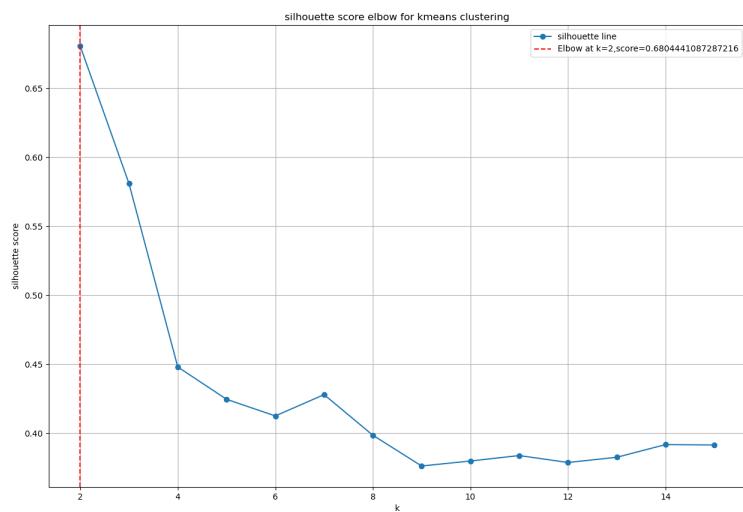
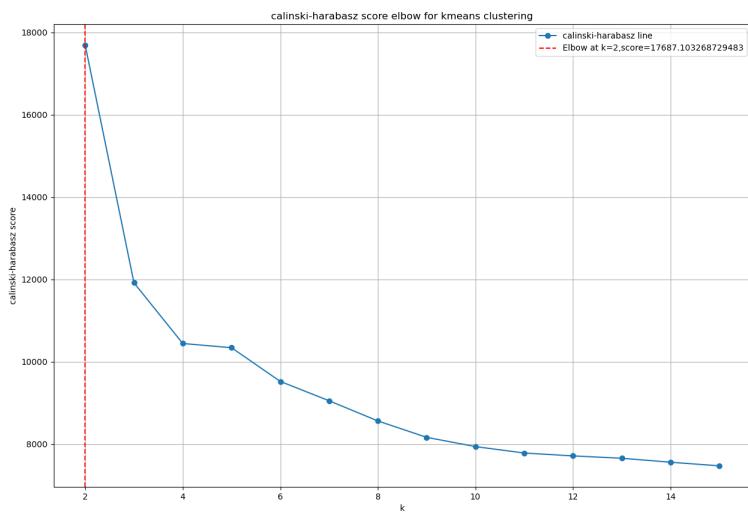
4.3. Klaszterezési algoritmusok

Klaszterezéshez több fajta algoritmust teszteltünk. A legjobb eredményeket az OPTICS (Ordering Points To Identify the Clustering Structure) [3] algoritmus adta. Aminek az eredményei a fenti képeken látható (lásd. 16). A képen bal oldalon láthatók rendre a KMeans, DBSCAN és BIRCH által rendezett klaszterek, a jobb oldalon pedig az OPTICS által rendezett klaszterek. Az utolsó sorban a BIRCH eredménye látható, ahol két klasztert egybevont, amit az OPTICS meg tudott különböztetni. OPTICS-on kívül teszteltük a KMeans, DBSCAN és BIRCH algoritmusokat.

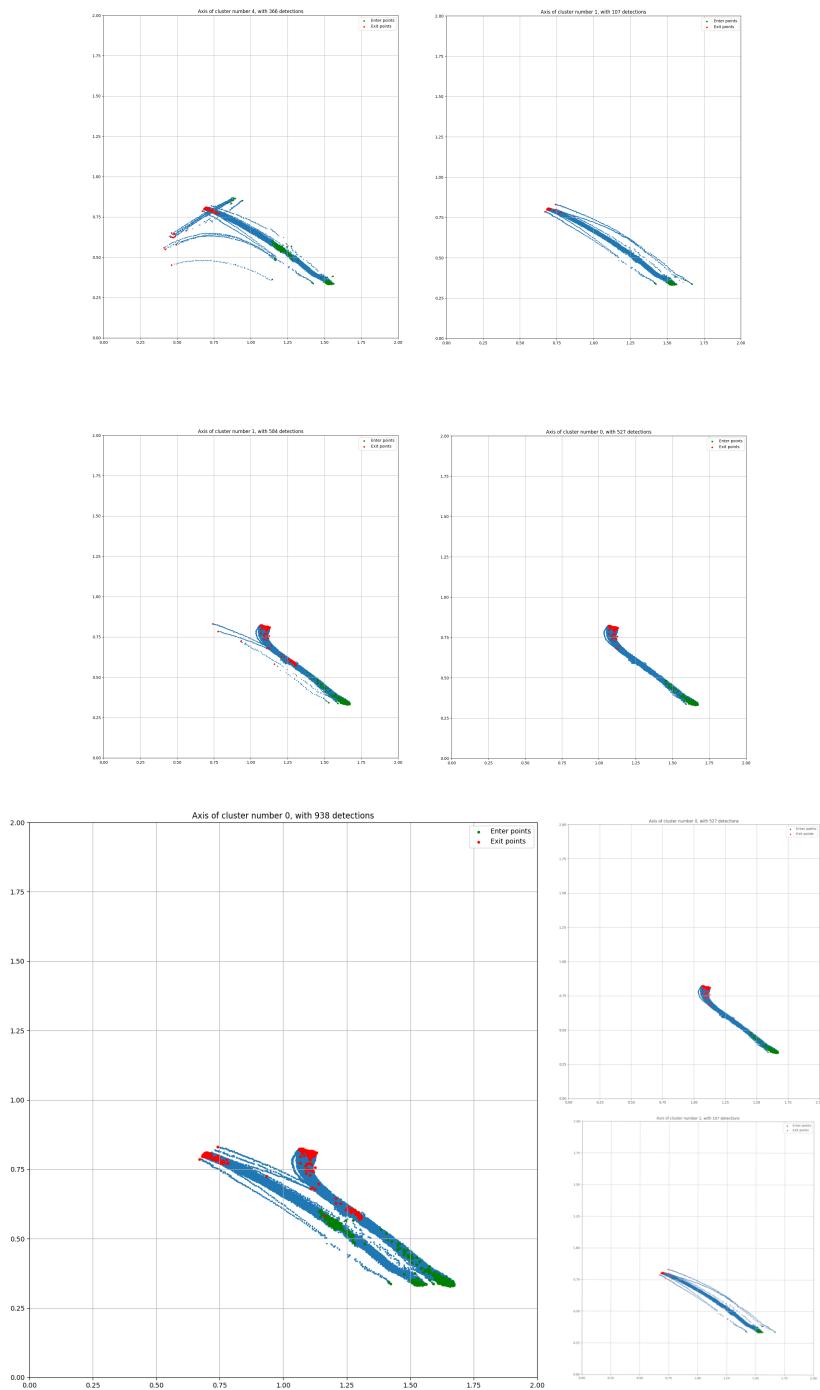
KMeans A KMeans klaszterezés egy unsupervised machine learning algoritmus, amely célja az adatok csoportosítása oly módon, hogy azonos klaszterbe tartozó adatok közötti távolság minimális legyen, míg az eltérő klaszterek közötti távolság maximális. Az algoritmus működése a következő lépésekkel áll:

1. Centroidok inicializálása: Az algoritmus véletlenszerűen inicializál k centroidot a dataseten, ahol k a klaszterek száma.
2. Adatok csoportosítása: Az algoritmus minden adatponthoz hozzárendeli a legközelebbi centroidet, és azonos klaszterbe helyezi azokat az adatpontokat, amelyeknek a centroidja megegyezik.
3. Centroidok újraszámolása: Az algoritmus újraszámolja a centroidok pozícióját 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 előírja, 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 találhatók 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 [19], Calinski-Harabasz Index [8] és Davies-Bouldin Index [10]. Elbow diagramok segítségével próbáltuk eldönteni, hogy milyen értéket érdemes adni az $n_clusters$ paraméternek lásd 15. A mérőszámok konzisztensen alacsony értéket adtak, egy négyágú kereszteződésnél, ahol jóval több klaszterbe sorolhatók a trajektóriák. A KMeans használatát ezért elvetettük.



15. ábra. Elbow diagramok



16. ábra. KMeans, BIRCH, DBSCAN vs OPTICS

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 során 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 mennyiségű 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 bemeneteket 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 egybevont 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 azonosítsa, 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. Megtalálja 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. Megkeresi az összes szomszédos adatpontot, és kiszámítja a távolságot az adott adatponttól.
3. A szomszédos adatpontokat távolság és sűrűség szerint rednezi.
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 sok erőforrást 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 16. képen, a KMeans, BIRCH, DBSCAN klasztereit állítjuk szembe az OPTICS által rendezett klaszterekkel. Látható, hogy az OPTICS, nagyon hatékonyan tudta kiszűrni a zajos trajektóriákat, és nem vont egybe kimeneti vagy bemeneti klasztereket.

4.4. Paraméterek kiválasztása

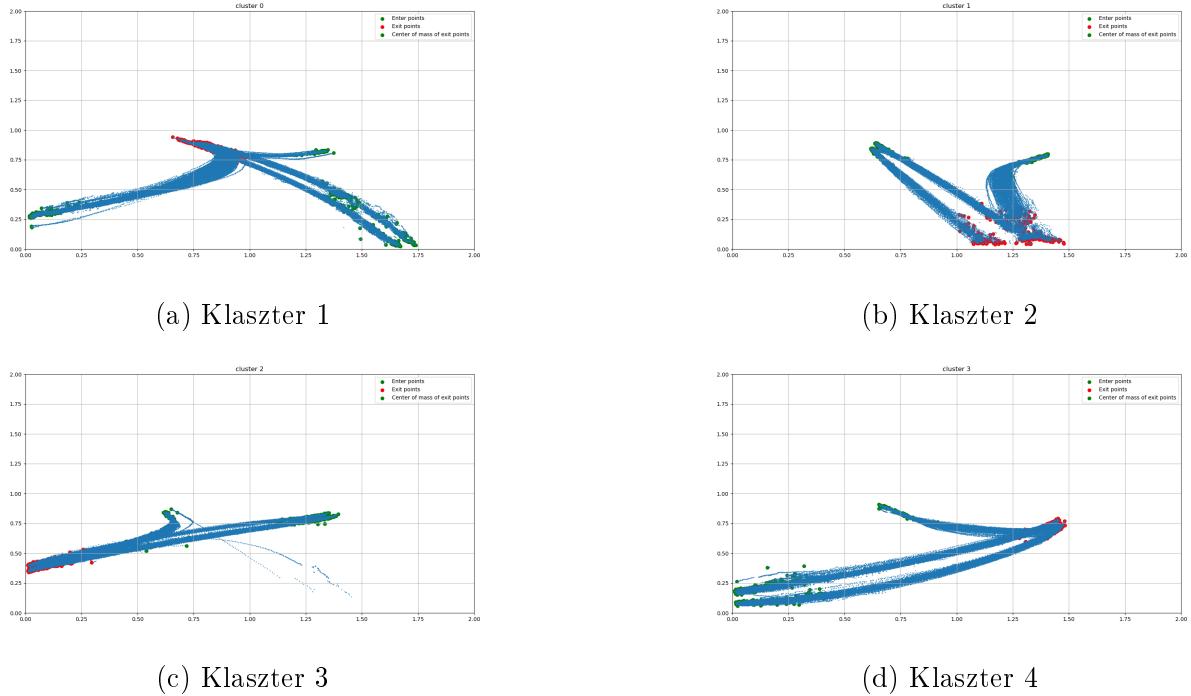
A megfelelő paraméterek kiválasztása a klaszterezéshez igen fonzosnak bizonyult. Ezt a gyűjtött adathalmazokon kézzel kellett finomhangolnunk. A halmazok minimum számososságát a *min_samples* paraméterrel lehet szabályozni, a pontok egymástól való távolságának felső határát *max_eps*-el lehet megadni. Az távolság kiszámítására használt metódust *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.

4.5. Közeli klaszterek egyesítése

Osztályozó algoritmusok tesztelése során azt figyeltük meg, hogy azt is hibának számítottuk amikor az osztályozó algoritmus két nagyon hasonló klaszter közül, amiknek a kimeneti koordinátáik azonosak, csak a bemeneti koordinátáik térnek el, nem a megfelelőbe sorolta az objektumot, viszont ha csak a kimeneti koordináták alapján pontozunk, akkor nem számít hibának. Erre a problémára a kimeneti koordináták alapján való csoportosítás a megoldás (lásd 17). Az egész klaszterezési eljárást nem kell megismételni, hanem a meglévő klaszterekre alkalmaztunk klaszterezést. Erre a KMeans algoritmust használtuk. Az algoritmus az eredeti és az összevont klaszter középpontok euklideszi távolságának minimalizálásán alapul. Az algoritmus implementációja pythonban:

```
def kmeans_mse_clustering(
    X: np.ndarray, Y: np.ndarray,
    n_jobs: int = 10, mse_threshold: float = 0.5
) -> Tuple[np.ndarray, KMeans, int, float]:
    Y_exitcluster_centers = calc_cluster_centers(X, Y)
    mse = 9999
    n_clusters = 1
    aoi_labels_best = []
    n_clusters_best = n_clusters
    clr_best = None
    # run min search
    while (mse > mse_threshold):
        clr = KMeans(n_clusters).fit(Y_exitcluster_centers)
        aoi_labels = clr.labels_
        cluster_centers = clr.cluster_centers_
        # calculate mse of original clusters and new pooled clusters
        distance_vector = [euclidean_distance(
            cluster_centers[1, 0],
            Y_exitcluster_centers[i, 0],
            cluster_centers[1, 1],
            Y_exitcluster_centers[i, 1])
            for i, l in enumerate(aoi_labels)]
        # if new mse is lower than the previous one, save the new one
        if np.max(distance_vector) < mse:
            mse = np.max(distance_vector)
            n_clusters_best = n_clusters
            clr_best = clr
            aoi_labels_best = aoi_labels
        n_clusters += 1
    Y_reduced_labels = np.zeros(shape=Y.shape, dtype=np.int32)
    for i, l in tqdm.tqdm(enumerate(Y), desc="Reduce labels"):
        Y_reduced_labels[i] = aoi_labels_best[l]
    return (
        Y_reduced_labels, clr_best, n_clusters_best, mse, aoi_labels_best)
```

A képeken zölddel jelöltük a bemeneti pontokat és pirossal a kimeneti pontokat. Látható az algoritmus eredménye, a különböző belépési de azonos kilépési pontú útvonalakat egybevonta, ezzel új klasztereket létrehozva.



17. ábra. Összevont klaszterek

5. Osztályozás

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 célkitűzésektől és az adatok jellegétől függ. A Multiclass klasszifikáció kevesebb osztály számánál jó eredményt adhat, a mi esetünkben 10-15 osztály is lehet, ami azt jelenti, hogy nem lehet elérni nagy pontosságot. Ennyi osztály közül nehéz pontosan eltalálni, 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 jól meghatározva.

5.4. Machine Learning modellek

Kutatásunk során több féle machine learning modellt teszteltünk: KNN (KNearesNeighbors), GNB (GaussianNaiveBayes), MLP (MultiLayerPerceptron), SGD (StochasticGradientDescent), SVM/SVC (SupportVectorMachine/SupportVectorClassifier) [9], DT (DecisionTree) [6]. Ezekből a GNB, 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.

1. táblázat. Bellevue Newport V1

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%

2. táblázat. Testset 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%

3. táblázat. Cross-Validation 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%

4. táblázat. Testset V7 Stride 15

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%

5. táblázat. Testset 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%

6. táblázat. Cross-Validation V7 Stride 15

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%

7. táblázat. Cross-Validation 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%

8. táblázat. Clusters (OPTICS)

	Top-1	Top-2	Top-3	Balanced Accuracy
KNN	95.69%	99.01%	99.76%	92.14%
SVM	86.45%	96.52%	98.94%	76.57%
DT	95.61%	97.05%	97.40%	92.68%

9. táblázat. Pooled clusters (K-Means MSE Search)

	Top-1	Top-2	Top-3	Balanced Accuracy
KNN	98.74%	99.79%	99.79%	98.73%
SVM	92.93%	98.83%	99.93%	93.14%
DT	96.92%	97.32%	97.32%	97.03%

KMeans csoportösszevonás Az általunk létrehozott csoportösszevonó algoritmus azért volt szükséges, mert a fentebb megjelenített pontosság értékek megtéveszthetők lehetnek. Mivel az útvonalak bemeneti és kimeneti pontok alapján vannak elsősorban csoportosítva, így az a predikció is hibának számít ami az eredeti csoporthoz közeli csoportba sorolta a járművet. Ezt nem feltétlenül kell hibának néznünk, mert nekünk a kimeneti pontot kell figyelembe vennünk, ami lehet ugyan az két csoportnál, amiknek különböző a bemeneti pontjuk (lásd 17). Az összevonás előtti és utáni eredmények megtekinthetők a 8 és 9 táblázatokban, amik a Bellevue NE adathalmazon futtatott kiértékelésből származnak. Az eredmények alapján látható, hogy a csoportösszevonás javította a predikciókat, a Top-1 és Top-2 pontosságok nőttek, a Balanced Accuracy pedig 5-10%-al javult a KNN és SVM esetében.

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 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 manhattani 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 osztálycímeknek 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, nemlineáris 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 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 pozíójuk 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 outlier-ek 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.

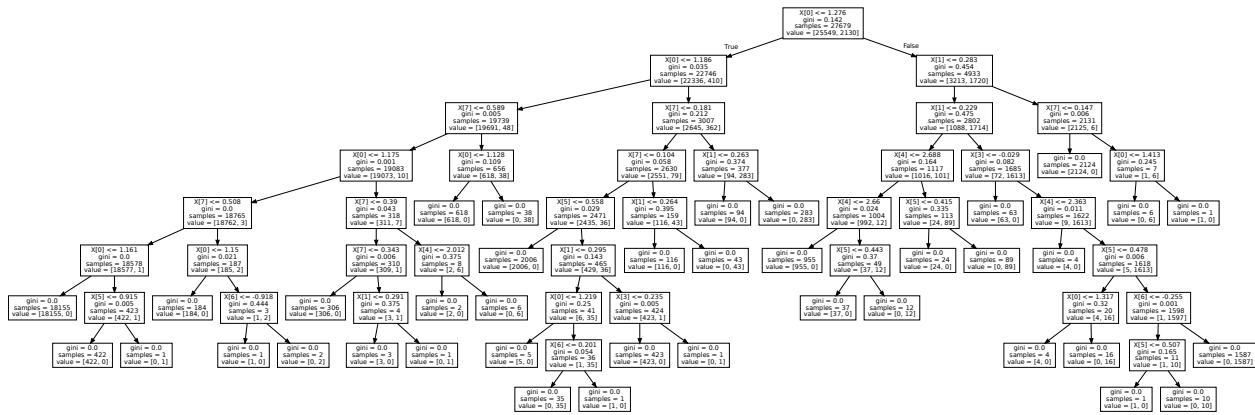
SVM Paraméterek Többféle "kernel function" - szűrő - közül választhatunk, ami lehet lineáris, polinomiális, exponenciális stb. Mi az RBF (Radial Basis Function) exponenciális szűrőt használtuk, aminek két fő paramétere van: C és γ . C az SVM egy általános paramétere, ami másik szűrők használatakor is jelen van. Ez a paraméter határozza meg mennyire legyen

elsimítva az osztályok közötti határ. Alacsony C sima határvonalhoz vezet, amíg a magas C azt okozza, hogy minden tanító adatot pontosan osztályozni tudjon. A γ prarméter, azt határozza meg, hogy egy tanító adatnak mekkora befolyása van. Magas γ esetén az egymáshoz közelebbi adatok magas befolyással bírnak egymásra. Méréseink 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 levélcsomópontokban pedig a végleges osztálycímek találhatók. A döntési fában minden ág egy adott tulajdonság értékét reprezentálja, és a fa 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 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 18). Nagy és komplex fák alakulhatnak ki, amik túltanulást eredményezhetnek, és már kis eltérések a tanító halmazban nagy eltéréseket eredményezhetnek a fa logikájában.



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 trajektóriákat, 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 trajektóriák ennél a buffernél több detektálást tartalmaznak, ezért az első verzióról a trajektóriát k részre osztottuk, így egy trajektória szelet mérete $s_n = n_d/k$, ahol n_d a detektálások számossága a trajektóriában. Ezekből a szeletekből képeztük az egyes feature vektorokat.

A hetedik verzió Felépítése $[x_0 \cdot w_1, y_0 \cdot w_2, v_{x_0} \cdot w_3, v_{y_0} \cdot w_4, x_l \cdot w_5, y_l \cdot w_6, v_{x_l} \cdot w_7, v_{y_l} \cdot w_8]$. 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 eredményéből 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 pontosságán (lásd 5). Még nagyobb 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 trajektóriából több feature vektort állítunk elő. Ezeknek a számosságát, a feature vektorokat generáló algoritmusban szabtuk meg. Ezzel azt is szabályoztuk, hogy mekkora időszeletből prediktáljon a modellünk. Mint ahogy fent is említettük, valós időben 15, max 30 detektálást érdemes tárolni a bufferben. A mérii eredmények azt mutatják, hogy 15 és a 30 nagyságú buffer között nincs nagy különbség pontosságból. 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 hozzátartozó valódi érték, akkor az eltalált predikciók és összes predikció hányadosa, 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 azért 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 valódi 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 perdikció az i . mintának, így a balanced accuracy-t így definiálhatjuk:

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

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

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

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 adatahalmazt 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 - Kereszt validáció

A túltanítás elkerülése érdekében, alkalmaztunk egy elterjedt metódust a cross-validationról. A cross validation 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 megméri 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-validationról származó eredmények a 3., 6 és 7. táblázatban mutatjuk be.

5.6.3. Teszthalmazos validáció

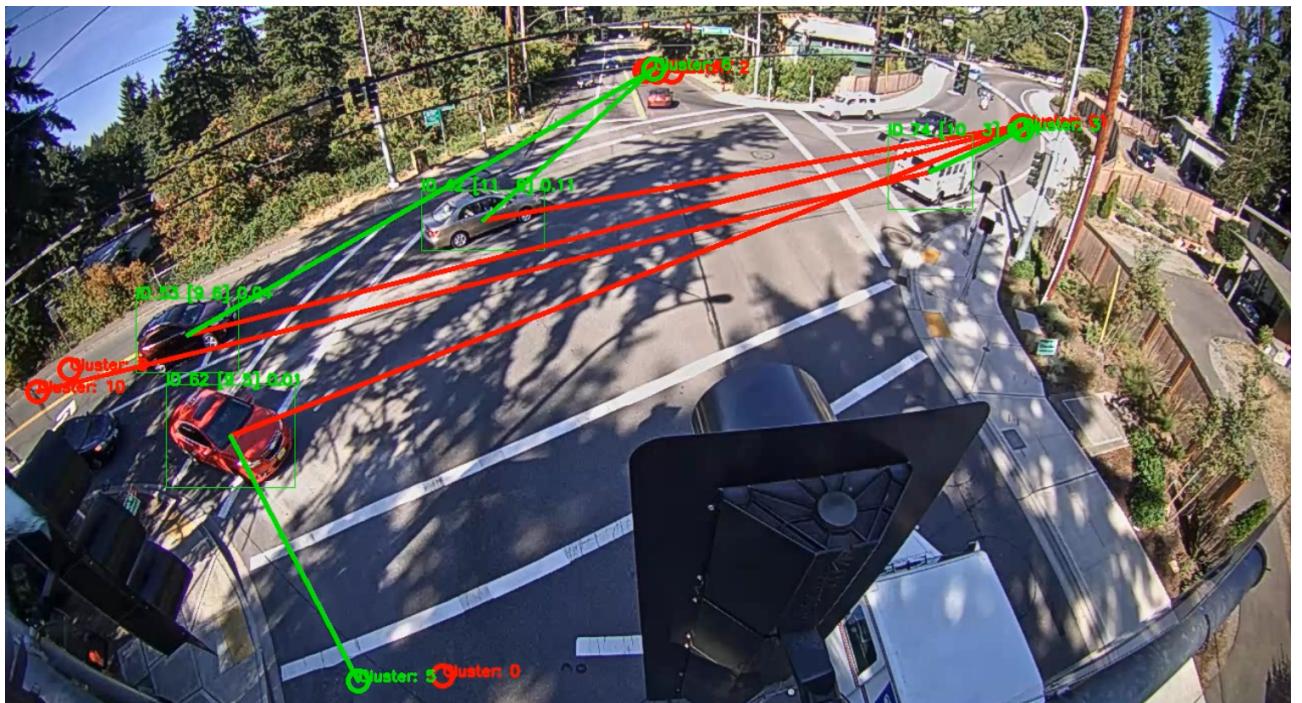
Hogy meggyőződjünk arról, hogy biztosan nem tanítottuk túl a modellünket, egy olyan teszt adathalmazon is le kell tesztelnünk, amit nem használtunk fel egyszer sem cross-validation alatt tanításra. Ezzel a méréssel bebizonyosodhatunk róla, hogy a modellünkben nincsen bias. A teszthalmazos mérés eredményeit a 2., 4 és 5. táblázatban mutatjuk be.

5.6.4. Modellek tárolása

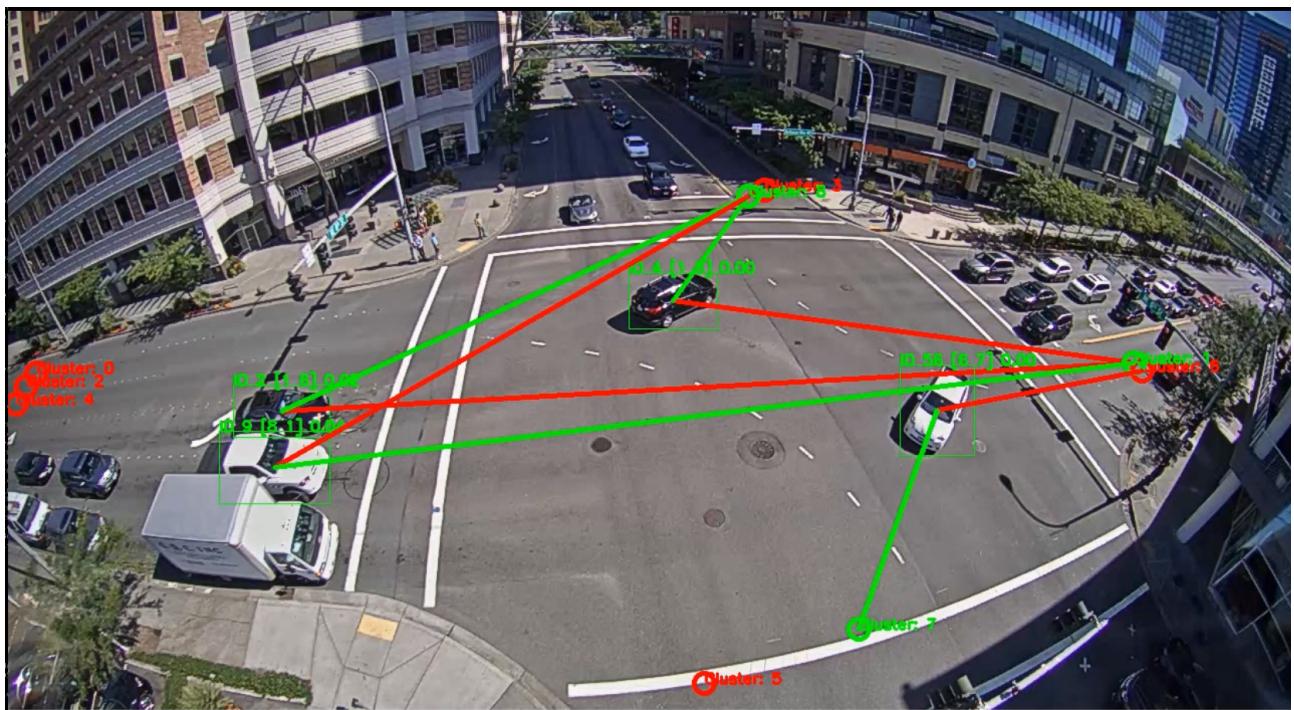
A betanított modelleket joblib file-ként tároltam el, amit később python-nal tudunk betölteni. Azért döntöttem a joblib mellett, mert a scikit-learn könyvtár is ezt használja, és ezt ajánlják modeljeik eltárolásához.

6. Valós idejű alkalmazás

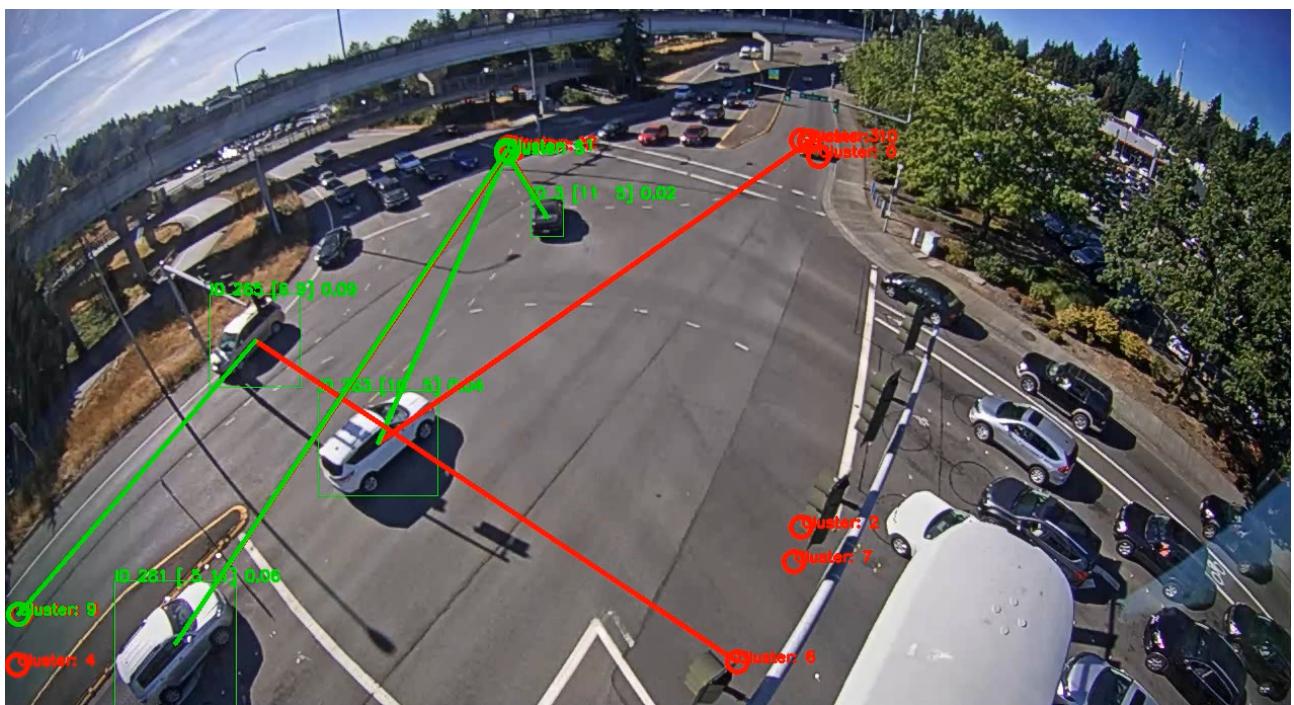
A modellek pontosságának tesztelésére nem csak mérőszámokat alkalmaztunk, hanem egy vizualizációs alkalmazást is fejlesztettem. Az alkalmazásnak meg kell adni a joblib modell fájlt és a videót, ami adatbázisul szolgált. Ezeken kívül meg lehet adni mekkora detekció buffert használjon és hogy a top mennyi predikciót rajzolja ki. A lejátszó kirajzolja a klaszterek kimeneti pontjait, és az autók középpontjával köti össze. A legvalószínűbb predikció zölddel, a kevésbé valószínű pedig pirossal van kirajzolva (lásd 19 20 22 21). Az alkalmazás futását a mellékletben megadott videókon lehet megtekinteni.



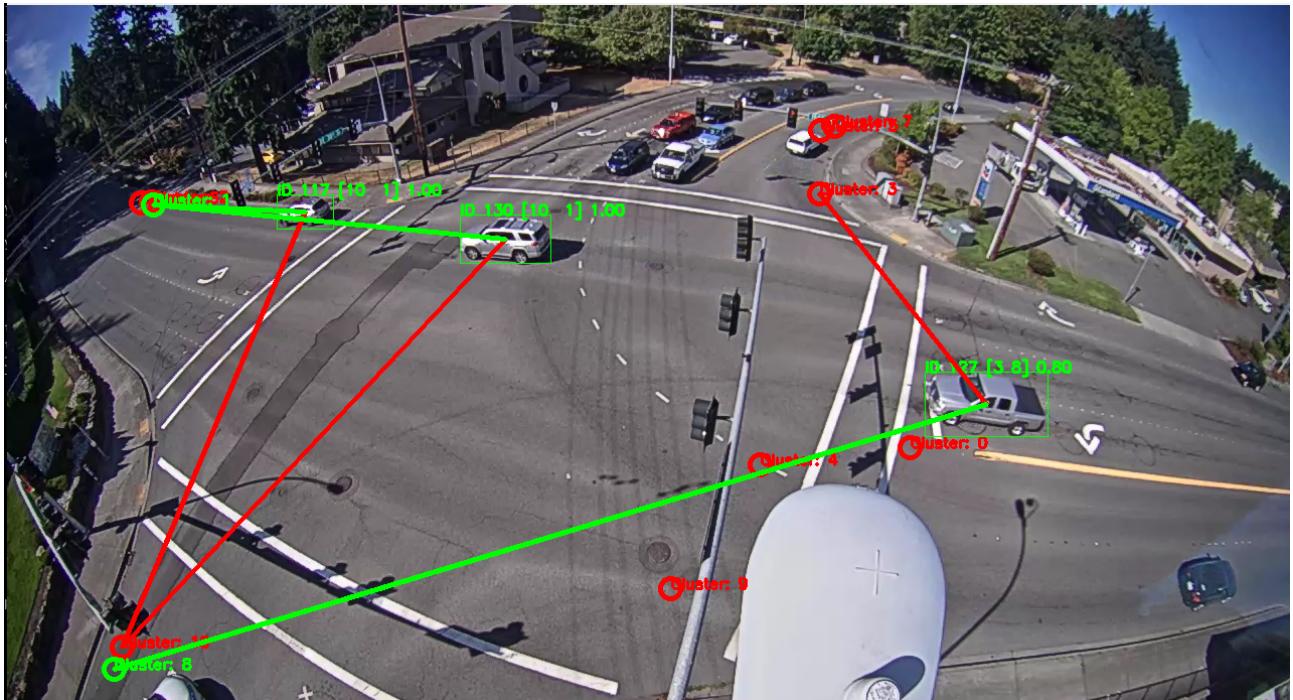
19. ábra. Bellevue Newport real time application



20. ábra. Bellevue NE real time application



21. ábra. Bellevue Eastgate real time application



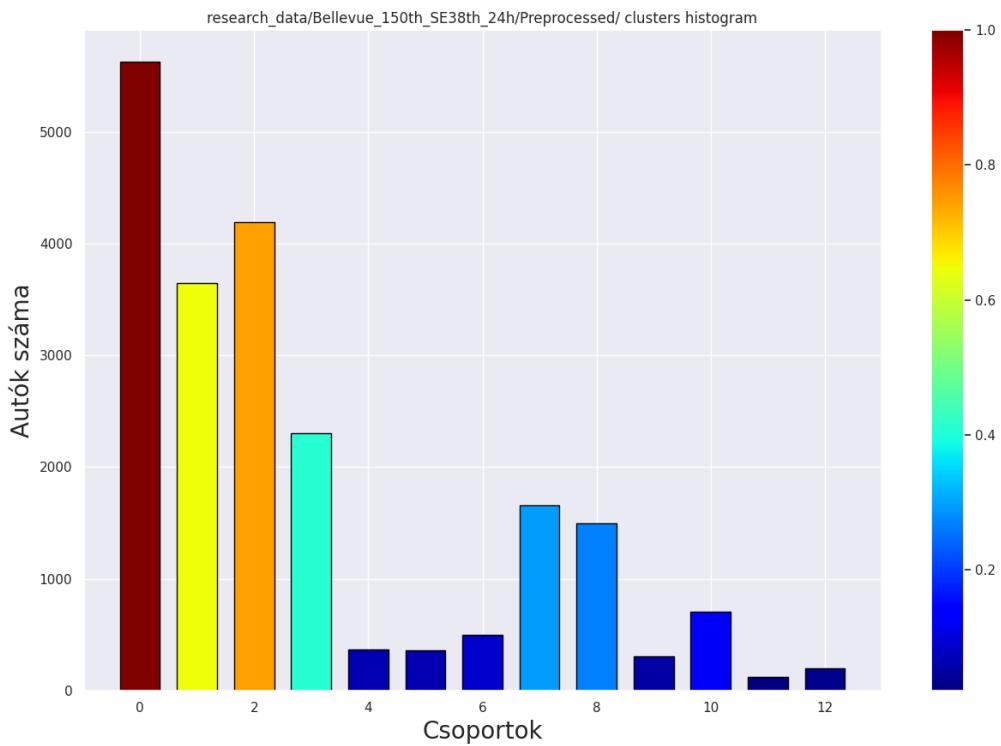
22. ábra. Bellevue SE real time application

7. Forgalmi statisztikák

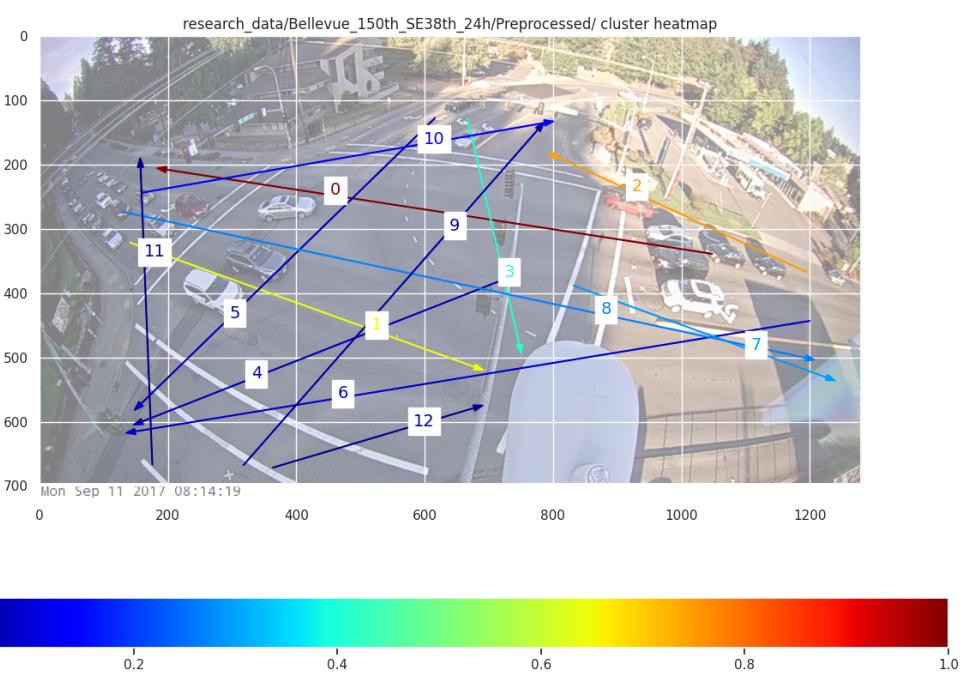
A kidolgozott klaszterezési technikákkal kellő finomsággal tudjuk felosztani az útvonalakat, amik alapján statisztikai adatokat tudunk kinyerni. A kinyert adatok alapján hiszogramokat és hőterképeket készítettünk. A hiszogramokon az egyes útvonalak teljes forgalmát tudjuk megjeleníteni, a hőterképeken pedig az egyes útvonalakon az autók óránkénti eloszlását. A hiszogramokat és hőterképeket a 23 24 25 26 27 képeken lehet megtekinteni. Forgalmi statisztikát generáló alkalmazásunkat részletesebben tárgyaljuk publikációnkban [1].

7.1. Hiszogram

A hiszogrammal az egyes csoportok teljes forgalmát tudjuk szemléltetni (lásd 23). Hogy a hiszogramot könnyebb legyen értelmezni, a kereszteződés képére generáltuk az útvonalakat a hiszogram színeivel (lásd 26).



23. ábra. Hisztogram



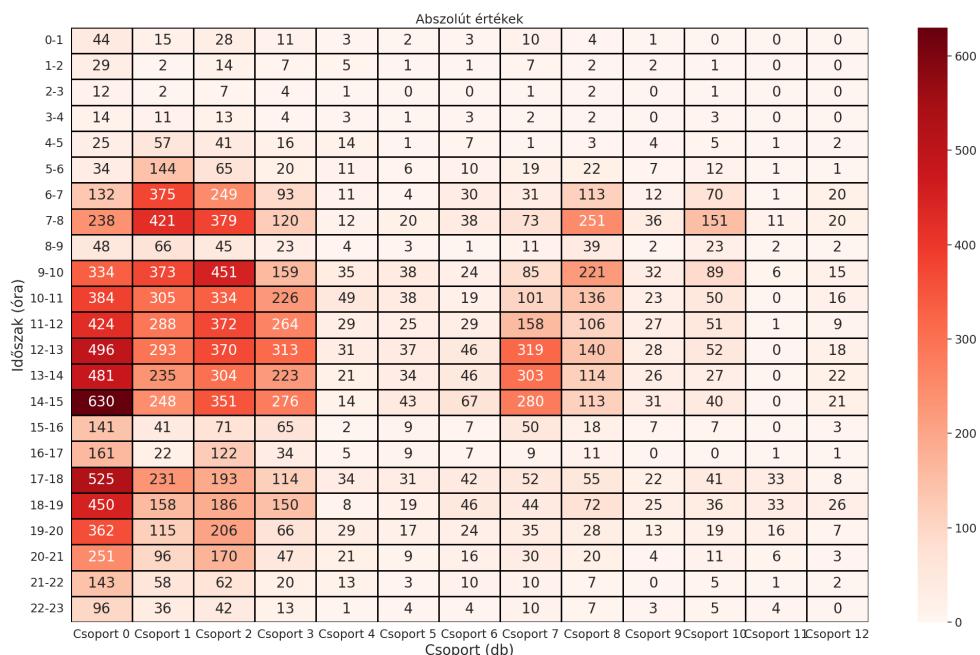
24. ábra. Hisztogram vizualizáció

7.2. Hőtérkép

Hőtérképen óránkénti és csoportonkénti felbontásban jelenítjük meg a járművek számát. A hőtérképeket a 25 26 27 képeken lehet megtekinteni.

Abszolút értékes

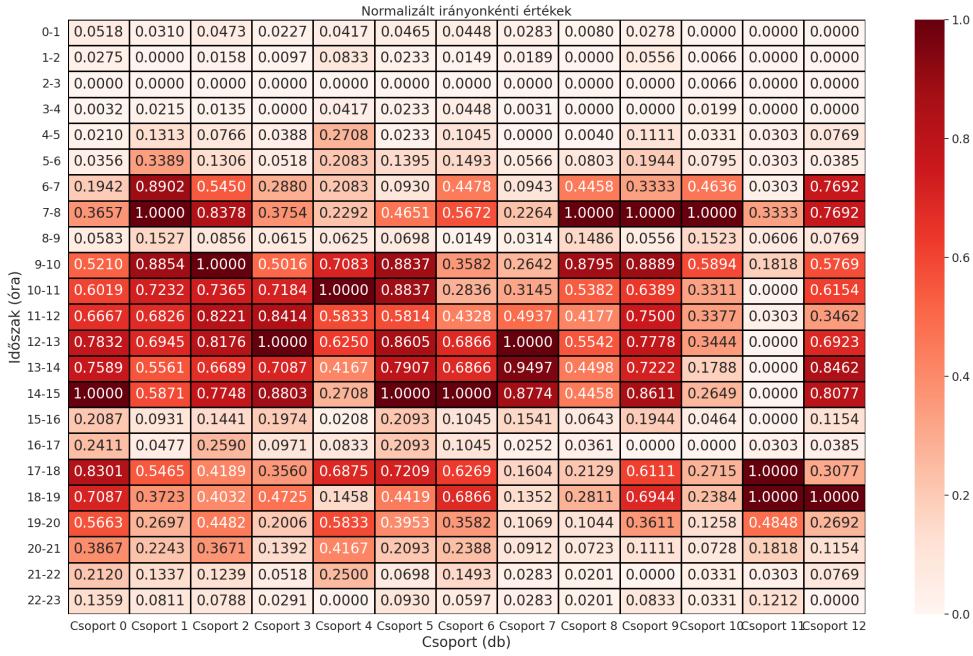
Több fajta reprezentációt választottunk, az egyik az abszolút értékes megjelenítés, ahol az elhaladó járművek tényleges számát jelenítjük meg. Itt a színezés az összes órát és útvonalat figyelembe veszi (lásd 25).



25. ábra. Abszolút értékes hőtérkép

Normalizált

A második reprezentációban a csoportonkénti normalizált értékeket jelenítjük meg, ahol az egyes csoportokban az óránkénti járművek számát normalizáljuk a csoporton belüli maximummal (lásd 26).



26. ábra. Klaszterenként Normalizált hőtérkép

Az utolso verzióban az óránkénti normalizált értékeket jelenítjük meg, ahol az egyes csoportokban az óránkénti járművek számát normalizáljuk az óránkénti maximummal (lásd 27).



27. ábra. Óránként Normalizált hőtérkép

8. Konklúzió

A kiépített keretrendszer a klaszterezéssel és klasszifikációs modellek betanításával egy fontos lépés ezen terület fejlesztésében.

A mérések jó alapot szolgálnak a jövőbeli kutatásoknak, milyen algoritmusokat érdemes még mélyebben megvizsgálni és melyekkel nem érdemes a továbbiakban foglalkozni. Más objektumdetektáló és objektumkövető algoritmusokat is érdemes lehet kipróbálni, amivel az adatgyűjtési fázist és az adattisztítási fázist lehet felgyorsítani és hatékonyabbá tenni.

Az átlagos 90% pontosság, amit elértünk a tanított modellekkel, nem tűnik túl magasnak, de figyelembe kell vennünk, hogy egy autó mozgása során sok olyan időszakasz van, amikor lehetetlen előre megmondani, melyik útvonalat fogja választani, mert pl. csak közelít a kereszteződéshez és nem választott még sávot. Ilyenkor az autó mozgása még semmilyen szinten nem utal a későbbi pályára, ezért óhatatlanul tévedések következnek be. A módszer predikciós képességét ilyenkor a "top-2 accuracy" mutatja: a tipikusan 10-15 lehetséges útvonal-csoport közül a választott 2 legvalószínűbb már 95% feletti esélyteljesen tartalmazza a ténylegesen bekövetkezőt.

Az adathalmaz növelésével a pontosságot jövőben még növelni lehet. A feature vektorok dimenziószámának növelésével és másfajta súlyozással 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 keretrendszerünkre is egy nagy előrelépés lehet, ezt úgy lehetne megvalósítani, hogy valós idejű futás közben, a bejövő adatokon nem csak osztályozást végzünk hanem ezeket az adatokat folyamatosan mentjük, majd ha elég adat összegyűlt időközönként frissítjük a modellt, így a modell adaptálódni tud a forgalom változásához.

A predikciós eljárásokon túl a statisztikai kimenetek hasznos információként szolgálhatnak a forgalom megértéséhez, és a városi közlekedés optimalizálásához. Ezeket a kimeneteket legfőképp közlekedés mérnökök tudják felhasználni.

Hivatkozások

- [1] A. Horváth Á. Agg B. G. Péter. „Automata forgalmi statisztika objektumdetektálás és adaptív járműtrajektória klaszterezés alapján”. *Közlekedés és Mobilitás* 2:1 (elfogadott). in press, software available at https://github.com/Pecneb/computer_vision_research.
- [2] D. Anguita és tsai. „The 'K' in K-fold Cross Validation”. *The European Symposium on Artificial Neural Networks*. 2012.
- [3] Mihael Ankerst és tsai. „OPTICS: Ordering Points to Identify the Clustering Structure”. *SIGMOD Rec.* 28.2 (1999. jún.), 49–60. old. ISSN: 0163-5808. DOI: 10.1145/304181.304187. URL: <https://doi.org/10.1145/304181.304187>.
- [4] G. Bradski. „The OpenCV Library”. *Dr. Dobb's Journal of Software Tools* (2000).
- [5] Georg Brandl. „Sphinx documentation”. URL <http://sphinx-doc.org/sphinx.pdf> (2021).
- [6] L. Breiman és tsai. „Classification and Regression Trees”. 1984.
- [7] Kay Henning Brodersen és tsai. „The Balanced Accuracy and Its Posterior Distribution”. *Proceedings of the 2010 20th International Conference on Pattern Recognition*. ICPR '10. USA: IEEE Computer Society, 2010, 3121–3124. old. ISBN: 9780769541099. DOI: 10.1109/ICPR.2010.764. URL: <https://doi.org/10.1109/ICPR.2010.764>.
- [8] Tadeusz Caliński és Harabasz JA. „A Dendrite Method for Cluster Analysis”. *Communications in Statistics - Theory and Methods* 3 (1974. jan.), 1–27. old. DOI: 10.1080/03610927408827101.
- [9] Chih-Chung Chang és Chih-Jen Lin. „LIBSVM: A library for support vector machines”. *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>, 27:1–27:27.
- [10] David L. Davies és Donald W. Bouldin. „A Cluster Separation Measure”. *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-1.2 (1979), 224–227. old. ISSN: 1939-3539. DOI: 10.1109/TPAMI.1979.4766909.
- [11] Martin Ester és tsai. „A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise”. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. KDD'96. Portland, Oregon: AAAI Press, 1996, 226–231. old.
- [12] Charles R. Harris és tsai. „Array programming with NumPy”. *Nature* 585.7825 (2020. szept.), 357–362. old. DOI: 10.1038/s41586-020-2649-2. URL: <https://doi.org/10.1038/s41586-020-2649-2>.
- [13] Richard D Hipp. *SQLite*. 3.31.1. verzió. 2020. URL: <https://www.sqlite.org/index.html>.
- [14] J. D. Hunter. „Matplotlib: A 2D graphics environment”. *Computing in Science & Engineering* 9.3 (2007), 90–95. old. DOI: 10.1109/MCSE.2007.55.
- [15] Joblib Development Team. *Joblib: running Python functions as pipeline jobs*. 2020. URL: <https://joblib.readthedocs.io/>.
- [16] Anand Paul és tsai. „Chapter 8 - Big Data collision analysis framework”. *Intelligent Vehicular Networks and Communications*. Szerk. Anand Paul és tsai. Elsevier, 2017, 177–184. old. ISBN: 978-0-12-809266-8. DOI: <https://doi.org/10.1016/B978-0-12-809266-8.00008-9>. URL: <https://www.sciencedirect.com/science/article/pii/B9780128092668000089>.

- [17] F. Pedregosa és tsai. „Scikit-learn: Machine Learning in Python”. *Journal of Machine Learning Research* 12 (2011), 2825–2830. old.
- [18] Luca Rossi és tsai. „Vehicle trajectory prediction and generation using LSTM models and GANs”. *PLOS ONE* 16.7 (2021. júl.), 1–28. old. DOI: 10.1371/journal.pone.0253868. URL: <https://doi.org/10.1371/journal.pone.0253868>.
- [19] Peter J. Rousseeuw. „Silhouettes: A graphical aid to the interpretation and validation of cluster analysis”. *Journal of Computational and Applied Mathematics* 20 (1987), 53–65. old. ISSN: 0377-0427. DOI: [https://doi.org/10.1016/0377-0427\(87\)90125-7](https://doi.org/10.1016/0377-0427(87)90125-7). URL: <https://www.sciencedirect.com/science/article/pii/0377042787901257>.
- [20] Erich Schubert és tsai. „DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN”. *ACM Trans. Database Syst.* 42.3 (2017. júl.). ISSN: 0362-5915. DOI: 10.1145/3068335. URL: <https://doi.org/10.1145/3068335>.
- [21] The pandas development team. *pandas-dev/pandas: Pandas*. latest verzió. 2020. febr. DOI: 10.5281/zenodo.3509134. URL: <https://doi.org/10.5281/zenodo.3509134>.
- [22] Chien-Yao Wang, Alexey Bochkovskiy és Hong-Yuan Mark Liao. „YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors”. *arXiv preprint arXiv:2207.02696* (2022).
- [23] Nicolai Wojke és Alex Bewley. „Deep Cosine Metric Learning for Person Re-identification”. *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)*. IEEE. 2018, 748–756. old. DOI: 10.1109/WACV.2018.00087.
- [24] Tian Zhang, Raghu Ramakrishnan és Miron Livny. „BIRCH: An Efficient Data Clustering Method for Very Large Databases”. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. SIGMOD '96. Montreal, Quebec, Canada: Association for Computing Machinery, 1996, 103–114. old. ISBN: 0897917944. DOI: 10.1145/233269.233324. URL: <https://doi.org/10.1145/233269.233324>.