

Szegedi Tudományegyetem
Informatikai Intézet

SZAKDOLGOZAT

Zsurzsa Dóra

2024

**Szegedi Tudományegyetem
Informatikai Intézet**

**Képmódosítás detektálása
hatékony SVM osztályozással**

Szakdolgozat

Készítette:
Zsurzsa Dóra
Programtervező
informatikus BSc szakos
hallgató

Témavezető:
Dr. Németh Gábor
egyetemi adjunktus

Szeged
2024

Feladatkiírás

A témát kiíró oktató neve: Dr. Németh Gábor

A témát meghirdető tanszék: Képfeldolgozás és Számítógépes Grafika Tanszék

Típus: Szakdolgozat

Kik jelentkezhetnek: 1fő, Programtervező informatikus BSc szakos hallgató

A feladat rövid leírása: A digitális képmódosítások detektálására nagyon sokféle algoritmus létezik. Sreeletha és szerzőtársai kontrasztjavítás után vizsgálták a blokkokra számolt hisztogramok csúcsait és hézagait, amelyek alapján SVM osztályozóval jelölik meg a vélhetően módosított képrészleteket. A hallgató feladata Sreeletha és szerzőtársai "Tampered Region Detection on Digital Images by Efficient SVM Classifier" című cikkében közölt algoritmus implementálása tetszőleges programozási nyelven, valamint az algoritmus kiértékelése nyilvános képi adatbázison.

Előismeret: nem szükséges

Szakirodalom:

"Tampered Region Detection on Digital Images by Efficient SVM Classifier ",
International Journal of Emerging Technologies and Innovative Research (www.jetir.org),
ISSN:2349-5162, Vol.2, Issue 10, page no.36-42, October-2015,
Available :<http://www.jetir.org/papers/JETIR1510007.pdf>

Tartalmi összefoglaló

- ***A téma megnevezése:***

Képmódosítás detektálása hatékony SVM osztályozással.

- ***A megadott feladat megfogalmazása:***

A feladat egy olyan program elkészítése, amely egy kapott képről eldönti, hogy korábban az módosítva volt-e, és ha igen, akkor megmutatja mely területek voltak manipulálva.

- ***A megoldási mód:***

A feladatot a főként a [3] és másodlagosan a [4] számú forrásokban leírt szakirodalmak alapján kerültek implementálásra. A program először a képet kisebb, egyenlő méretű, egymást nem átfedő blokkokra bontja, majd minden blokkra kiszámítja a hisztogram alapú jellemzőket. Ezeket a jellemzőket egy SVM osztályozóval elemezzük, amely meghatározza, hogy a kép egyes területei módosítva voltak-e. Végül a módosított blokkokat megjelenítjük a felhasználónak.

- ***Alkalmazott eszközök, módszerek:***

A fejlesztés Pycharm programmal történt Python nyelven.

- ***Elért eredmények:***

Az implementált algoritmus sikeresen detektálta a kontrasztjavítással módosított régiókat a tesztelt képeken.

- ***Kulcsszavak:***

digitális képmódosítás, kontrasztjavítás, hisztogram elemzés, SVM, képmódosítás detektálása

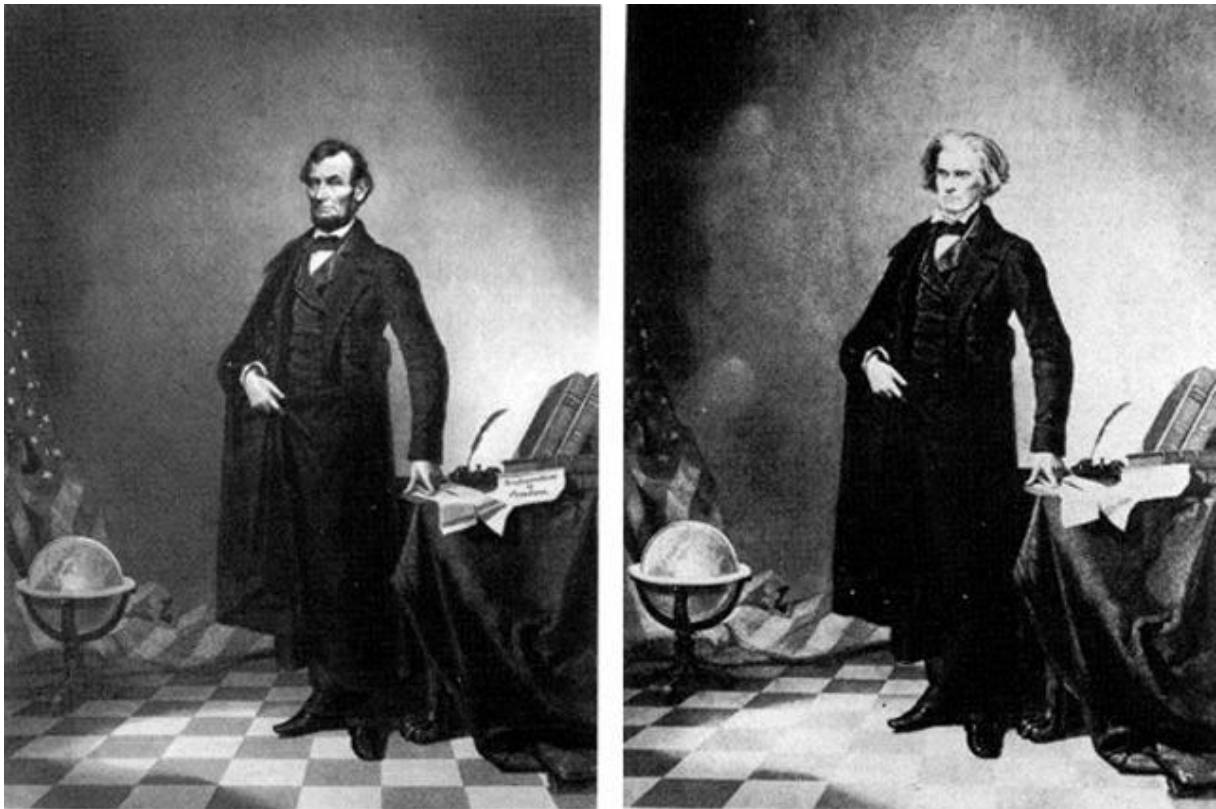
Tartalomjegyzék

Feladatkiírás	2
Tartalmi összefoglaló	3
Tartalomjegyzék.....	4
1. BEVEZETÉS	5
2. AZ ALGORITMUS ISMERTETÉSE	8
2.1. Globális kontrasztjavítás detektálás	8
2.2. Lokális kontrasztjavítás detektálás	10
3. IMPLEMENTÁCIÓ	11
3.1. Blokkonkénti peak/gap tartományok pozíciója.....	11
3.2. Futam alapú hasonlóság mérés	17
3.3. Csúcs alapú hasonlóság mérés	20
4. SVM	21
4.1. SVM jellemzők előállítása	21
4.2. Osztályozás nem lineáris SVM-mel	25
5. EREDMÉNYEK.....	29
6. A PROGRAM HASZNÁLATA	30
IRODALOMJEGYZÉK	31
Nyilatkozat	32
Mellékletek	33

1. BEVEZETÉS

A digitális képek manipulációja egyre elterjedtebbé vált internet uralta korszakunkban. Ezt a technológiát a katonaság, a rendőrség és az orvosok is széles körben használják mindennapi tevékenységeik elvégzésekor. A képszerkesztő eszközök gyors fejlődésével akár már egy egyszerű átlagember is képes fotók manipulációjára. Ennek súlyos következményei is lehetnek, például egy rosszindulatú személy könnyűszerrel terjeszthet álhíreket vagy csaló reklámokat.

Az első híressé vált képmanipuláció az 1860 körül készült Lincoln-Calhoun kompozit volt. Az ikonikus portrén Abraham Lincoln feje egy volt alelnök, John C. Calhoun fényképére lett illesztve. A hamisításra azért volt szükség, mert a Lincoln elleni merénylet előtt nem készült az elnökről egyetlen megragadó kép sem, ami egy vezetőhöz méltó lenne. Az arcát kedvenc fotográfusa, Mathew Brady által korábban készített ülő portréjáról vették, és ráhelyezték Calhoun egyik régebbi fényképére.



1.1. ábra: Lincoln-Calhoun kompozit [1]

Aktuális példaként említeném meg Katalin hercegnéről 2024. március elején megjelent fotót, melyet több nemzetközi fotóügynökségnek is vissza kellett vonnia manipuláció lehetősége miatt. Az eredeti posztot az [1] forrásban találhatjuk meg. Az anyák napján közzétett

fotó alatt a hercegné köszönetet mond azokért a jókívánságokért, amelyeket az elmúlt két hónapban kapott korábbi műtétjével kapcsolatban. Ha alaposan megnézzük a képet, akkor viszont észrevehető jó pár részlet, ami azt mutatja, hogy a fotó manipulálva volt. Például a ruhadarabok részei sok helyen nem illenek össze, a hercegné jobb keze és haja homályos a háttérrel együtt, ráadásul a lépcsők is szaggatottnak tűnnek. Nem sokkal a kép visszavonása után a hercegné bocsánatot is kért a Twitter nevezetű közösségi oldalon, amiben úgy fogalmazott, hogy mint sok más amatőr fényképész, ő is néha kísérletezik képek szerkesztésével. További részleteket a [2] forrásban találhatunk.



1.2. ábra: Katalin hercegné anyák napi fotója [2.a]

A képmanipulálásnak sok különböző módszere létezik. Az egyik legismertebb például a retusálás, amivel a képeken hibákat javíthatunk ki vagy távolíthatunk el, például bőrhibákat tüntethetünk el, haját, fogakat tökéletesíthetünk. Montázsokkal különböző képeket vagy képrészleteket rakhatunk össze úgy, hogy egy új kép jöjjön létre. Például, ha nem tetszik az

egyik képnél a háttér, akkor kicserélhetjük egy erdőre. Színkorrekciók és szűrők használatával pedig egy kép színvilágán és hangulatán tudunk javítani. Ehhez a kép kontrasztját javítjuk, aminek detektálásával az algoritmusunk fog foglalkozni.

Az új Polgári Törvénykönyv (Ptk.) 2013-as bevezetése óta a személyiségi jogok védelme kiterjed a képmáshoz és hangfelvételhez tartozó jogokra is. A Ptk.-t tartalmazó oldal az [5] forrás *b* részében található. A Ptk. második könyvének XI. címében található meg az alábbi rendelkezések. Minden személynek joga van a nevéhez, képmásához és hangfelvételéhez, amelyeket a (2:42. §) paragrafus általánosan védelmez. Fontos kiemelni, hogy a felvételek és képek felhasználásához szükséges az érintett személy hozzájárulása, kivéve, ha a felvétel nyilvános közszereplés során készült és a felvétel/kép felhasználása nem sérti az érintett jogait. Az egyedi szabályok a képmáshoz és hangfelvételhez való jogok tekintetében részletesen meghatározzák, hogy mikor és milyen körülmények között lehetséges a felvétel készítése és felhasználása hozzájárulás nélkül. Erről a (2:48. §) paragrafus tesz említést. Ha nézünk egy példát, mondjuk egy koncerten készítünk felvételt, amiben nem fókuszálunk egy konkrét egyénre, akkor a személy hozzájárulása nem feltétlenül szükséges. A jogi gyakorlat tovább részletezi ezeket a jogokat és szabályokat. Rejtett kamerával készült felvételek elfogadhatósága a szabad bizonyítás elvének alapján ítéltető meg, de figyelembe kell venni, hogy a felvétel nem sérti-e valakinek a személyi jogait. A jogi gyakorlatban manipulált képek esetén a bíróságoknak minden esetben egyedileg kell meghatározniuk, hogy a manipulált képek használata vagy terjesztése hogyan érinti a személyiségi jogokat és az alkotói szabadságot, illetve ennek megfelelően dönteniük kell az esetleges jogi következményekről.

A szakdolgozatomban egy olyan program elkészítése volt a cél, amely segíthet a kontrasztjavítással manipulált képek felismerésében. A képfeldolgozás mindig is érdekes téma volt számomra és a nemrégiben teljesített Digitális képfeldolgozás kurzus is ezt a tudásvágyat erősítette bennem. Első lépésként tájékozódtam, hogy a témával kapcsolatban milyen kutatásokat végeztek. Két főbb elméleti kidolgozás keltette fel a figyelmemet, melyekről a későbbiekben részletesebben is szót ejtek. Ezután következett a program megvalósításához szükséges fejlesztői környezet és a programozási nyelv kiválasztása. Én a Pycharm fejlesztői környezetet és a Python nyelvet választottam. Mivel a szoftver nyílt forráskódú, van egy óriási

közössége, aminek a segítségével néhány csomag telepítése után könnyen tudunk képfeldolgozási műveleteket végrehajtani.

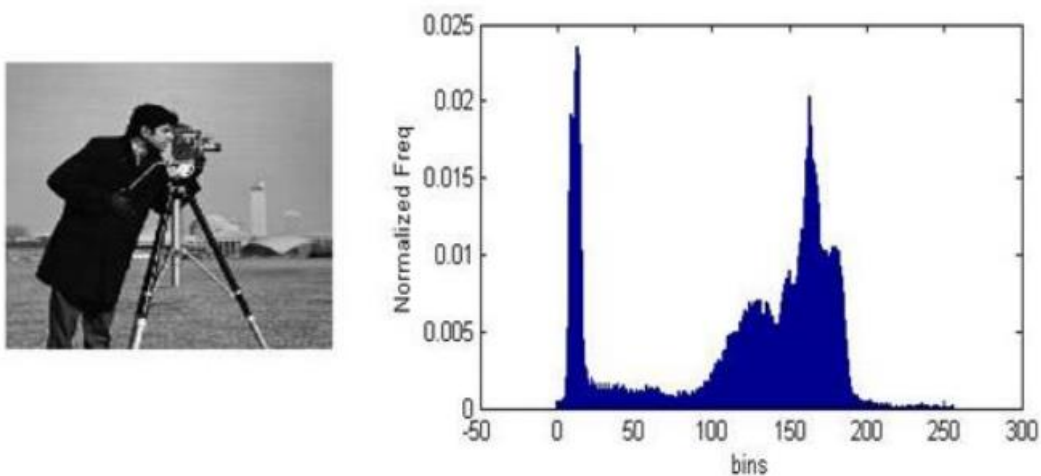
2. AZ ALGORITMUS ISMERTETÉSE

A digitális képek manipulációjának felismerése a képhamisítás és adathamisítás felderítésében kulcsfontosságú. Az algoritmus célja, hogy hatékonyan detektálja a digitális képek kontrasztjavítás során módosított régióit lokálisan és globálisan is. A feladat kivitelezése a [3] forrás alapján történt, és a továbbiakban egy rövid áttekintést olvashatunk a főbb lépésekről. Először is az algoritmusnak globálisan kell detektálnia, hogy történt-e kontrasztjavítás a képen, mivel, ha egy kép teljes egészében kontrasztjavításon esett át, és nem csak egy bizonyos része, akkor ez globális kontrasztjavításra utalhat. A globális kontrasztjavítás detektálásával lehetőségünk lesz egy általános mintát vagy elrendezést felismernünk, amin később finomíthatunk a lokális kontrasztjavítás detektálásnál. Ennél a detektálásnál részletesebb információkat nyerhetünk ki a kontrasztjavítások elhelyezkedéséről és méretéről egy képen belül. Ezen felül segíthet megkülönböztetni a képen belül olyan részeket a kontrasztjavítástól, amelyek nem manipulált hatások során kerültek a képre. A globális és lokális detektálási módszer együttes használata segít javítani a detektálás pontosságát és hatékonyságát. Végző soron pedig az SVM osztályozó segítségével a manipulált régiókat tudjuk azonosítani. Az osztályozó három részből áll. Első a tanítási fázis, itt az ismert információk alapján egy osztályozót hozunk létre. A második a tesztelési fázis, itt az ismeretlen tartalmat osztályozzuk a tanított modell segítségével. Végül pedig a csúcs és futam hasonlóságok alapján a kép manipulált régióit detektálja. Ezeknek az alapvető lépéseknek a segítségével hatékonyan és megbízhatóan fel tudjuk ismerni a digitális képeken végrehajtott kontrasztjavítást.

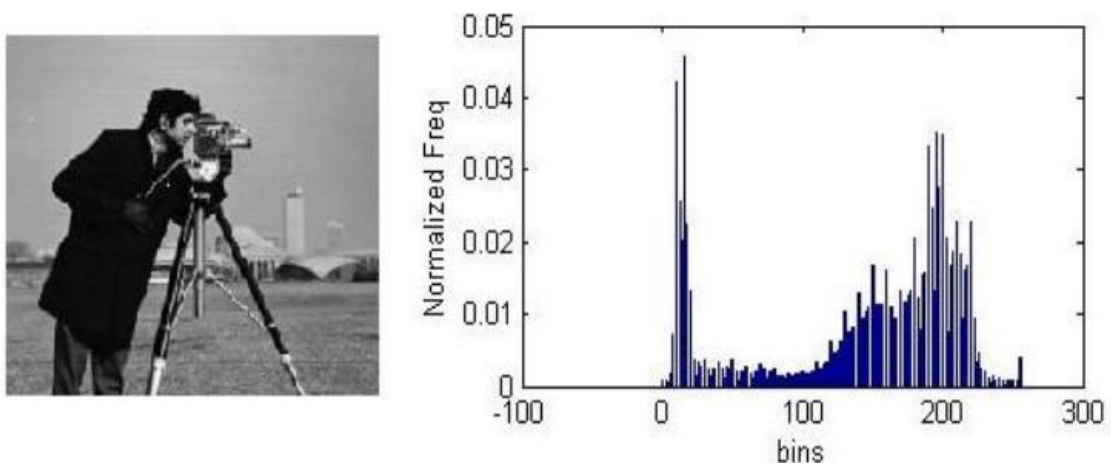
2.1. Globális kontrasztjavítás detektálás

A digitális képek kontrasztjavítása általában azt jelenti, hogy a pixelértékeket egy új értéktartományba képezzük le, hogy vizuálisan vonzóbb eredményt érjünk el. Az implementált algoritmus feltételezi, hogy az eredeti, azaz nem módosított képek hisztogramjai simaságot mutatnak, amit az **2.1 ábra** szemléltet, míg a kontrasztjavított képek hisztogramjai csúcsokat (peak) és futamokat (gap) tartalmaznak, ahogy a **2.2. ábrán** is látszik. Mindkét ábra a [3] forrásból származik. Ezek a futamok nulla magasságúak a hisztogramokon, és mindig megjelennek módosított képeken. Az ilyen típusú futamoknál két típust kell megkülönböztetnünk. Az első típusba olyan nulla magasságú futamokat tartoznak, amiknek a

szomszédjaiban létezhet ugyanilyen futam. A második típusba pedig olyan nulla magasságú futamok tartoznak, amelyeknek szomszédságában természetellenes módon nincs egyetlen nulla magasságú futam sem, ezeket **izolált hézag tartományoknak** nevezzük és erre a típusra fog fókuszálni az algoritmus. Az algoritmusnak a globális kontrasztjavítás észlelés része megszámlálja a **hézag tartományokat** az input kép hisztogramjában. Ezután összehasonlítja ezt a számot a döntési küszöbvel. Ha a szám nagyobb a küszöbnél, akkor a kontrasztjavítás észlelve lesz, ha az érték alacsonyabb a küszöbnél, akkor pedig nem történik detektálás. A döntési küszöböt tapasztalati úton vagy gépi tanulással határozhatjuk meg.



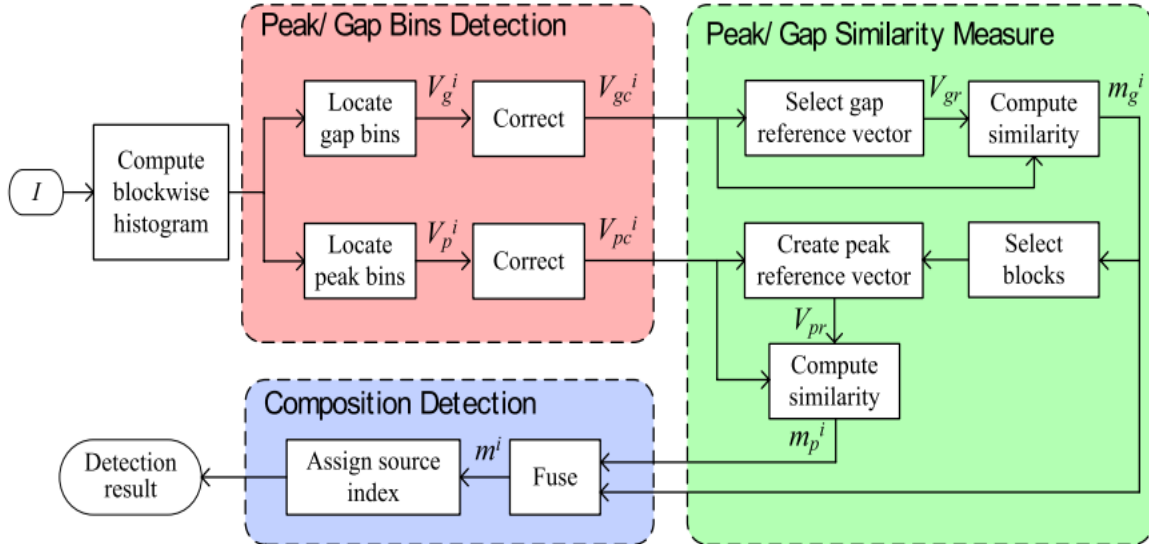
2.1 ábra: Eredeti kép és hozzá tartozó hisztogram [3]



2.2. ábra: Manipulált kép és hozzá tartozó hisztogram hézag tartományokkal [3]

2.2. Lokális kontrasztjavítás detektálás

A képmásolással és beillesztéssel történő hamisításokat lokális kontrasztjavítás detektáló algoritmussal derítjük fel. Az alábbi ábrán látható a kompozit kép észlelő technika.



2.3. ábra: Kompozit kép észlelő technika [3]

Ahogy az az ábráról is leolvasható, az I input képet először felosztjuk egymást nem fedő azonos méretű blokkokra. Ezt követően blokkonként kiszámoljuk a hisztogramokat. A bin-ek hisztogramokban lévő tartományok, amelyekben adatokat elemzünk vagy csoportosítunk. A gap bin-ek olyan tartományokat jelölnek, amelyekben üres területeket találhatunk. Ezután megkeressük a futam tartományokat és a korábban leírt globális kontrasztjavítás detektáló módszerrel lementjük V_g^i változóba a tartományok pozícióit. A csúcs pozíciók meghatározásához kivonjuk az eredeti, futamokkal teli hisztogramot a szűrt a szomszédokkal feltöltött verziójából és ezután küszöböljük a hisztogram értékeket. Az ilyen módon megkapott futam és csúcs pozíciókat szintén lementjük egy változóba, ami ebben az esetben V_p^i lesz. Ahhoz, hogy tovább csökkentsük az érzékelési hibákat, a kapott csúcs és futam pozíciókat javítjuk úgy, hogy a legtöbb blokkban megtartjuk az egyszerre létezőket. Az ilyen futam pozíciókat V_g változóban lementjük. Ahhoz, hogy eltüntessük az olyan futam tartományokat, amik nem kontrasztjavítás során jöttek létre, a javított futam és csúcs pozíció vektort Hadamard szorzattal generáljuk.

A két típusú régió (eredeti és kontrasztjavított) megkülönböztetése érdekében először meghatározunk egy referencia pozíció-vektort. Minden blokkot osztályozni lehet a pozíció-vektor és a referencia pozíció-vektora közötti hasonlóság alapján. A blokk, amiben a legtöbb

hézag tartomány van, nagy valószínűséggel egy nem módosított régióban van. Ahhoz, hogy megmérjük a hasonlóságot a futam pozíció-vektorok között, minden futamot tartalmazó párt meg kell vizsgálni először. A hasonlóságot V_{gc}^i és V_{gr} között m_g^i fogja jelezni, amit az alábbi egyenlet definiál.

$$m_g^i = \frac{\sum_{k \in \Omega_i \cap \Omega_{gr}} V_{gc}^i(k) \cdot V_{gr}(k)}{\sum_{k \in \Omega_i \cap \Omega_{gr}} V_{gc}^i(k) \cdot V_{gr}(k) + \overline{V_{gc}^i}(k) \cdot V_{gr}(k) + V_{gc}^i(k) \cdot \overline{V_{gr}}(k)},$$

2.4. ábra: Hasonlóság kiszámítása V_{gc}^i és V_{gr} között [4]

A V_{gc}^i változó a javított futam pozíció és V_{gr} a futam referencia vektor. A referencia csúcs pozíció V_{pr} vektort úgy állítjuk elő, hogy egyesítjük azokat a csúcs pozíció vektorokat, amik nagyobb eséllyel jönnek V_{gr} régiójából. A hasonlóságot V_{gc}^i és V_{pr} között, azaz csúcsot tartalmazó párokat mentünk le ebbe az új változóba, amit m_p^i -el jelölünk, hasonlóan definiáljuk, azzal a különbséggel, hogy a futam változókat kicseréljük a megfelelő csúcsokkal. Ha nem létezik ilyen csúcsot tartalmazó pár, akkor m_p^i értéke 1 lesz.

3. IMPLEMENTÁCIÓ

A program fejlesztéséhez a Python programozási nyelvet választottam, mivel a nyelv könnyen érthető, és sok remek könyvtárral rendelkezik a képfeldolgozási feladatokhoz, a segítőkész közösség miatt.

3.1. Blokkonkénti peak/gap tartományok pozíciója

Az implementáció során először beolvassuk a bemeneti képet szürkeárnyaltos módban és ezt eltároljuk az **image** változóban.

```
1 image = cv2.imread('input_image.png', cv2.IMREAD_GRAYSCALE)
2 Nb = 32
3 blocks_original = [image[y:y + Nb, x:x + Nb] for y in range(0,
4 image.shape[0], Nb) for x in range(0, image.shape[1], Nb)]
```

Ehhez a **cv2** csomagot használjuk, aminek használata során először megadjuk a képfájlnak a nevét, amit be szeretnénk olvasni, majd a következő argumentumban pedig megmondjuk, hogy a képet szürkeárnyaltos módban olvassuk be.

Ezt követően a képet egyenlő méretű blokkokra osztjuk, ami jelen esetben 32 pixel, de lehet akár 8 vagy annak többszöröse is. A blokkok számát az Nb (number of blocks) változóban kerül eltárolásra.

A **blocks_original** változó pedig egy listakifejezést tartalmaz, amiben megtörténik a képnek a blokkokra osztása. Az `image[y:y + Nb, x:x + Nb]`-ben az argumentumok az y és x tengelyen a kivágandó rész kezdő- és végpozícióját jelölik, amihez azért adjuk hozzá az Nb változót, hogy meghatározzuk a kivágott blokk magasságát az első argumentumban, illetve a szélességét a második argumentumban, ezzel biztosítva, hogy mindegyik blokk ugyanakkora méretű legyen. Ezt követően végig iterálunk a tengelyeken, hogy megkapjuk a blokkok kezdő sorainak és oszlopainak indexeit. Az első for ciklussal a kép magasságán iterálunk végig y változóval és Nb lépésközzel, míg a másodikkal a kép szélességén iterálunk végig x változóval és Nb lépésközzel.

Ezután ismét inicializálunk két változót, amik a következő sorokban kerülnek felhasználásra. Ezek közé tartozik **coexisting_gap_positions_all_blocks**, ami a **zero_height_gap_bins_coordinates_per_block** változóval együtt egy üres lista. Nevükből adódóan mindkettő pozíciók tárolására fog szolgálni.

```
1 coexisting_gap_positions_all_blocks = []
2 zero_height_gap_bins_coordinates_per_block = []
3 hist = cv2.calcHist([image], [0], None, [256], [0, 255])
4 norm_hist = hist / hist.sum()
```

Először is a `cv2.calchist()` függvény segítségével kiszámítjuk a szürkeárnyaltos kép hisztogramját. Első paraméterként több képet is megadhatnánk, de nekünk csak az **image** változóban lévő képre van szükségünk. Ezt követi a csatornák listája, de mivel a szürkeárnyaltos képeknek csak egy csatornája van, ezért csak azt használjuk. Utána jön a maszk, ami jelen esetben nem szükséges, ezért `None` értéket adunk meg. Ezután megadjuk a hisztogram méretét mindegyik dimenzióban. A szürkeárnyaltos képeknél egy pixel intenzitása 0 és 255 közötti értéket vehet fel. A 0 jelöli a teljesen fekete színt, a 255 pedig a teljesen fehéret. Ennek az értéktartománynak a segítségével egy kép összes pixelének intenzitását megfelelően tudjuk reprezentálni egy hisztogramban. Ebben a tartományban 256 különböző érték van, ezért a hisztogram mérete 256 lesz. A mi esetünkben is 256 lesz a hisztogram mérete, mivel szürkeárnyaltos képet használunk és persze a tartomány is 0-tól 255-ig fog tartani. A kapott hisztogramban az indexek az intenzitásértékek lesznek, az értékek pedig ezek előfordulásának gyakoriságát jelölik. Ahhoz, hogy a normalizált hisztogramot megkapjuk a **norm_hist** változóban, az előbb kiszámolt **hist** változót el kell osztanunk a `hist.sum()` függvénnyel. A

függvény a hisztogram összes intenzitásérték előfordulási gyakoriságának összegét adja meg, azaz kiszámolja az összegét.

Ezután felveszünk újabb változókat. Az első a **zero_height_gap_bins_db** nevű lista lesz, amit arra fogunk használni, hogy ahol a normalizált hisztogram nulla értéket vesz fel, eltároljuk azoknak az intenzitás értékeknek az indexeit. A második pedig a **zero_height_gap_bins** változó lesz, ebben egy olyan tömböt tárolunk, melynek mérete megegyezik blokkok számával és minden blokkhoz rendel egy 256 hosszú tömböt, amelyben 1-es értékkel fogjuk jelölni, hogy a blokkban az adott intenzitás értékhez tartozik **hézag tartomány**. Ha pedig nincs ilyen tartomány, azokat az eseteket 0-s értékkel fogjuk jelölni. Az *np.zeros()* függvény segítségével tudunk létrehozni egy olyan tömböt, aminek minden eleme nullával van inicializálva. A tömb elemeit azért készítjük elő ilyen módon, mert meg szeretnénk adni, hogy a tömb milyen hosszú legyen, azt viszont nem lehet megtenni anélkül, hogy konkrét értékeket helyeznénk bele, ezért csak nullákkal töltjük fel a tömböt először. Az első argumentumban van megadva a tömb mérete, a *len(blocks_original)* értéke adja meg a tömb sorainak számát, a 256 pedig az oszlopok számát jelöli, mivel az intenzitásértékek tartománya 0-tól 255-ig terjed. A második argumentum a tömb elemeinek típusát adja meg, ami jelen esetben egész számok lesznek.

Ezt követően egy ciklust futtatunk végig a normalizált hisztogram elemein.

```

1 for k in range(0, 255):
2     if norm_hist[k] == 0 and (min(norm_hist[k - 1], norm_hist[k + 1]) >
3 tau or (1 / (2 * w1 + 1)) * np.sum(norm_hist[k - w1:k + w1 + 1]) > tau):
4         zero_height_gap_bins_db.append(k)
5         zero_height_gap_bins[:, k] = 1
6 print("zero height:", zero_height_gap_bins)
7 print("Zero Height Gap Bins:", zero_height_gap_bins_db)

```

A ciklus minden *k* indexre megvizsgálja, hogy a *norm_hist[k]* értéke nulla-e és emellett teljesül az a feltétel is, hogy az adott index előtt és utána lévő értékek közül a kisebb érték nagyobb, mint a τ . Ezek mellett ellenőrizzük, hogy a *norm_hist[k - w1:k + w1 + 1]* intervallum összegének normalizált értéke nagyobb-e, mint a tau. Ha a megadott három feltétel közül legalább az első kettő teljesül, akkor a *k* indexet hozzáadjuk a **zero_height_gap_bins_db** listához, és a **zero_height_gap_bins** tömb megfelelő oszlopába beállítjuk az értéket egyre. Ezt követően a kapott eredményeket kiíratjuk a konzolra a könnyebb átláthatóság érdekében.

A biztonság kedvéért megszámoljuk a detektált hézag tartományok számát is.

```

1 Ng = len(zero_height_gap_bins_db)
2 print("Ng:", Ng)
3 # ellenőrzés döntési küszöb alapján
4 decision_threshold = 3
5 if Ng > decision_threshold:
6     print("Kontraszt javítás észlelve.")
7 else:
8     print("Kontraszt javítás nem található.")

```

Ehhez létrehozunk egy **Ng** nevű változót, amiben a *len()* függvény segítségével lekérjük a **zero_height_gap_bins_db** tömb hosszát és szintén kiíratjuk az adatot konzolra. Ezt követi egy ellenőrzés a döntési küszöb alapján. Ha az **Ng** változó értéke nagyobb, mint a küszöb értéke, akkor visszakapjuk konzolon azt az eredményt, hogy *Kontrasztjavítás észlelve*, ha pedig ez az érték nem üti meg a küszöböt, akkor pedig azt kapjuk vissza, hogy *Kontrasztjavítás nem található*.

A következő kódrészlet a blokkokban található hézag tartományok megkeresésére szolgál.

```

1 Vg = np.zeros((len(blocks_original), 256), dtype=int)
2 for i in range(len(blocks_original)):
3     block_hist_original = cv2.calcHist([blocks_original[i]], [0], None,
4     [256], [0, 256])
5     for k in zero_height_gap_bins_db:
6         if block_hist_original[k] == 0:
7             Vg[i, k] = 1
8 # 0 height gap binek számolása blokkonként
9 Ng_per_block = np.sum(Vg, axis=1)
10 print("Ng_per_block:", Ng_per_block)
11 print("Vg:", Vg)

```

A **Vg** változó egy **blocks_original** hosszúságú kétdimenziós tömböt hoz létre, ahol kezdetben minden elem értéke 0. Minden sor (vagyis blokk) egy egész hisztogramot fog képviselni, ahol az oszlopok száma 256 lesz, a 0-tól 255-ig terjedő tartomány miatt. Ezt követően egy for ciklusban, ami megegyezik a **blocks_original** hosszával (blokkok száma), egy *i* változóval minden blokkra kiszámítjuk az eredeti hisztogramot a *cv2.calcHist* függvény segítségével. Ebben beágyazva található egy másik for ciklus, amiben *k*-val végig iterálunk a **zero_height_gap_bins_db** változón és ha teljesül az a feltétel, hogy a változó adott indexén lévő érték nulla, akkor a *Vg[i, k]* értéket egyre állítjuk. Ezt követően megszámoljuk blokkonként a hézag tartományokat az **Ng_per_block** változó segítségével. Az eredmény kiszámításához a Numpy csomag *np.sum()* függvényét használjuk, ami hasonlóan működik a korábban látott *hist.sum()* függvényhez. Végezetül az **Ng_per_block** és **Vg** változókat kiíratjuk terminálba.

A következő kódrészlet a csúcsok pozícióinak meghatározását végzi el, amely a 3.4.3-as pontnak felel meg a [6] forrásban.

```

1 gap_filled_hist = np.zeros((len(blocks_original), 256), dtype=int)
2 for i in range(len(blocks_original)):
3     for k in zero_height_gap_bins_db:
4         if Vg[i, k] == 1: # üres hézag a blokkban
5             neighbors = []
6             for j in range(-w1, w1 + 1):
7                 if 0 <= k + j < 256:
8                     neighbors.append(block_hist_original[k + j])
9
10            # környezet átlagával feltölteni az üres hézagot
11            gap_filled_hist[i, k] = np.mean(neighbors)
12
13 # ellenőrzés, blokk esetén eredeti és gap-filled hisztogram
14 sample_block_index = 0
15 print("Original Histogram of Block", sample_block_index, ":",
16       cv2.calcHist([blocks_original[sample_block_index]], [0], None,
17 [256], [0, 256]).flatten())
18 print("Gap-filled Histogram of Block", sample_block_index, ":",
19       gap_filled_hist[sample_block_index])

```

Első lépésként egy **gap_filled_hist** nevezetű kétdimenziós tömböt hozunk létre, amelynek mérete megegyezik a **blocks_original** listában tárolt blokkok számával. Itt is minden blokkhoz tartozik egy hisztogram, amely a 256 különböző intenzitásérték gyakoriságát tárolja, továbbá minden elem értéke kezdetben 0 az *np.zeros()* nevű függvény használata miatt. Alatta két darab egymásba ágyazott for ciklust láthatunk. A külsőben minden blokkra, míg a belsőben minden blokkban található hézag tartományra iterálunk. Ha egy blokkban található egy hézag tartomány, akkor egyes értéket adunk át az adott indexnek, ezzel jelezve, hogy a blokkban üres hely van. A következő lépésben megvizsgáljuk a futam körüli szomszédos tartományokat. Ehhez létrehozunk egy **neighbors** nevű üres listát, amibe a szomszédos tartományok értékeit fogjuk gyűjteni. A szomszédos tartományok alatt a futam körüli **w1** szélességű ablakban található elemeket értjük. Ha az aktuális tartomány értéke 'k', akkor a szomszédos tartományok értékei **2*w1+1** és **k + w1** között találhatóak meg. Ezekre a szomszédos tartományokra azért van szükség, hogy a futamot körülvevő értékek közötti átlagot kiszámítsuk, és ezzel feltöltsük a futam tartományt. Végezetül kiírjuk az eredeti és a hézaggal feltöltött hisztogramot ellenőrzés céljából. Az utóbbi hisztogramot **gap_filled_hist** néven tároljuk el.

A második lépésben használatra kerül a *medfilt()* függvény.

```

1 filtered_version = medfilt(gap_filled_hist)

```

A függvény segítségével az előző lépésben kitöltött hisztogram alapján egy szűrt változatot hozunk létre. Maga a függvény egy medián szűrőt alkalmaz a bemeneti tömbre és egy másik tömböt ad eredményül, ahol az értékek mediánja az input tömb megfelelő része

alapján számítódik ki. A szűrés segítségével létrejött tömböt *filtered version* néven tároljuk el. A medián szűrés segít csökkenteni a zajt és kiemelni a jelentősebb strukturális részeket egy képen vagy hisztogramon.

A harmadik lépésben létrehozuk a V_p változót, amely segítségével tároljuk, hogy az egyes blokkokban és intenzitás értékekben található-e csúcs vagy sem.

```

1  Vp = np.zeros((len(blocks_original), 256), dtype=int)
2  for i in range(len(blocks_original)):
3      for k in zero_height_gap_bins_db:
4          if Vg[i, k] == 1: # Üres hézag a blokkban
5              # különbség számítása
6              diff = abs(gap_filled_hist[i, k] - filtered_version[i, k])
7              # csúcs detektálása küszöbérték alapján
8              if diff > tau:
9                  Vp[i, k] = 1
10 print("Vp:", Vp)

```

Minden blokkra és intenzitás értékre beállítjuk a V_p értékét nullára. Az első lépésben látott egymásba ágyazott for ciklusokat használjuk ismét, és megnézzük, hogy a V_g változóban található-e egyes érték. Ha igen, akkor a **diff** nevű változóban kiszámítjuk a különbséget a **gap_filled_hist**, és a **filtered_version** között. A számításhoz használjuk az *abs()* függvényt is, ezzel biztosítjuk, hogy a különbség mindig pozitív legyen. Ezek után megnézzük, hogy a különbség meghaladja-e a tau küszöbértéket. Ha igen, akkor a V_p tömbben az adott blokk és intenzitásérték pozíciójában beállítjuk az értéket egyre, jelezve, hogy csúcsot találtunk. Végül a V_p változót is kiíratjuk szemléltetés szempontjából.

A következő sorokban a korrigált futam pozíciókat fogjuk meghatározni.

```

1  k = 256
2  Vk = np.zeros(k)
3  Cg = np.zeros(k)
4
5  for k_index in range(256):
6      count = 0
7      for i in range(Nb):
8          if i * 256 + k_index < len(Vk) and Vk[i * 256 + k_index] == 1:
9              count += 1
10
11     if count > Nb / 2:
12         Vk[k_index] = 1
13     else:
14         Vk[k_index] = 0
15 print("VK:", Vk)
16 print("Cg:", Cg)
17 Vgc_i = Vg * Vk
18 print(Vgc_i)

```

A javításra azért van szükség, mert felléphetnek olyan esetek, amikor egyes futam tartományok nem kontrasztjavítás során jöttek létre. Először létrehozunk két üres 256 hosszú

tömböt **VK** és **Cg** néven. A tömbökben minden elemet nullával inicializálunk. Egy for ciklusban végig iterálunk az összes intenzitásértéken a **k_index** változóval és a **count** változó segítségével fogjuk megszámolni, hogy hány blokkban van jelen futam az adott intenzitásértékben. Ehhez segítségül veszünk egy újabb for ciklust, **i** változóval, ami 0-tól $Nb-1$ -ig fog tartani, és ennek segítségével számoljuk meg, hogy hány blokkban van futam az adott intenzitásértékben. Erre a feltételre azért van szükség, hogy a **k_index** ne lépje túl a tömb méretét. A beágyazott cikluson belül két feltételt kell ellenőriznünk. A $i * 256 + k_index < len(Vk)$ rész segítségével kiszámítjuk az adott blokkhoz és intenzitásértékhez tartozó indexet. Az első feltétel azt ellenőrzi, hogy az aktuális index ne lépje túl a **Vk** tömb hosszát, a második pedig azt nézi, hogy az adott indexhez tartozó elem értéke egyenlő-e egygel (azaz van-e futam az adott blokkban az adott intenzitásértéknél). Ha mindkét feltétel teljesül, akkor növeljük egygel a **count** változót, ami a futamok számát tárolja az adott intenzitásérték esetén. Az eredmények kiírása után jön a V_{gc}^i változó bevezetése, amiben a korrigált futam pozíciók lesznek tárolva. A változót a **Vg** és **Vk** változó szorzataként inicializáljuk és a kapott eredményt kiíratjuk terminálra.

3.2 Futam alapú hasonlóság mérés

A következő kódrészletben egy futam alapú hasonlósági mértéket fogunk definiálni. Ez arra szolgál, hogy meghatározzuk, melyik blokkban van a legtöbb futam, és ez alapján egy referenciavektort állít elő.

```

1  num_ones_per_block_i = np.sum(Vg, axis=1)
2  reference_block_index_i = np.argmax(num_ones_per_block_i)
3  Vgr_i = np.zeros_like(num_ones_per_block_i)
4  Vgr_i[reference_block_index_i] = 1
5  print("Referenciablokk indexe a Vg-ben:", reference_block_index_i)
6  print("Referenciapozíciós vektor (Vgr_i):", Vgr_i)
7  print("EDR a referenciapozíciós vektorhoz (Omega_gr):", Omega_gr)

```

A **num_ones_per_block_i** változóban fogjuk tárolni a blokkokban található futamok számát. A $np.sum(Vg, axis=1)$ függvény segítségével minden blokkra vonatkozóan összeszámoljuk a futamokat (azaz azokat az értékeket, ahol egyesek szerepelnek) a **Vg** tömbben. A **reference_block_index_i** változóban tároljuk a legtöbb futammal rendelkező blokk indexét. Ahhoz, hogy megkapjuk ezt az értéket, a Numpy csomag $np.argmax()$ függvényét hívjuk meg a **num_ones_per_block_i** változóra. Most következik a referencia vektor létrehozása, amit a V_{gr}^i nevű változóban fogunk létrehozni. Először is használni fogunk egy új függvényt a Numpy csomagból, aminek a neve $np.zeros_like()$. Ez a függvény egy olyan vektort hoz létre, aminek a mérete és alakja megegyezik a **num_ones_per_block_i** változóval és minden elemének

értéke nulla lesz. A $V_{gr_i}[reference_block_index_i] = 1$ kifejezés az jelenti, hogy a V_{gr}^i vektorban az az elem az értékét, amelynek indexe megegyezik a **reference_block_index_i** változó értékével, egyre állítjuk. Így a referenciavektorban csak egyetlen darab egyes lesz, ahol az a blokk található, amelyben a legtöbb futam található. Ezután szükségünk lesz egy **EDR**-re (Error Detection and Recovery) a referencia vektorhoz. Az **EDR**-eket hibák észlelésére és helyreállítására szokták használni. Az alábbi egy Ω_{gr} változóban fogjuk implementálni, amihez a Numpy csomag `np.arange()` függvényét használjuk. Ezzel a változó egy egytől 253-ig terjedő egész számokat tartalmazó vektort fog tartalmazni. A kapott eredményeket print függvénnyel kiíratjuk.

A következő részben a futam pozícióvektorok közötti megfeleltetést ellenőrizzük és kiszámoljuk az összesített hasonlóságot.

```

1  matched_pairs = 0
2  total_pairs = 0

3  for i in range(len(Vgr_i)):
4      for k in range(256):
5          if Vgr_i[i] == 1 and Vgc_i[i, k] == 1: # Case ①
6              matched_pairs += 1
7              total_pairs += 1
8          elif Vgr_i[i] == 0 and Vgc_i[i, k] == 1: # Case ②
9              total_pairs += 1
10         elif Vgr_i[i] == 1 and Vgc_i[i, k] == 0: # Case ③
11             total_pairs += 1
12 # hasonlóság kiszámítása
13 similarity = matched_pairs / total_pairs
14 print("Összesített hasonlóság V^i_gc és Vgr között:", similarity)

```

Két darab új változónk lesz, az első a **matched_pairs**, amiben eltároljuk, hogy hány olyan pár van, amelyben mind a referencia, mind a futam pozícióvektor egy adott indexénél szerepel egyes érték. Második változó pedig **total_pairs** lesz, itt összeszámoljuk, az összes olyan párt, ahol a futam pozícióvektorban található egyes érték. Két for ciklus segítségével végig megyünk először a V_{gr}^i vektoron i változóval, majd a futam pozícióvektoron k változóval, így minden referencia vektor eleméhez rendelünk megfelelő futam pozícióvektor elemeket és ellenőrizzük a megfeleltetést.

Ezt követően három esetet vizsgálunk. Első esetnél azt nézzük, hogy ha mind a referencia, mind a futam pozícióvektor egy adott indexénél szerepel egyes érték, akkor az azt jelenti, hogy a két vektor elemei megfelelnek egymásnak. Ha ez teljesül, akkor a **matched_pairs** és **total_pairs** változó értékét is növeljük eggyel. Második esetnél, ha a referencia vektorban nincs egyes érték, de a futam pozícióvektorban van, akkor az azt jelenti, hogy a futam pozícióvektorban lévő egyes érték egy olyan pozíciót jelöl, amely nem szerepel a referencia vektorban.

A feltétel teljesülésekor csak a **total_pairs** változó értékét növeljük eggyel. Az utolsó esetben pedig azt nézzük, hogy ha a referencia vektorban van egyes érték, de a futam pozícióvektorban nincs, akkor ez azt jelenti, hogy a referencia vektorban található egyes érték egy olyan pozíciót jelöl, amely a futam pozícióvektorban nem szerepel. Ilyenkor a **total_pairs** változó értékét növeljük eggyel. Mindezek után kiszámítjuk a hasonlóságot a **similarity** nevű változóban. A műveletet úgy végezzük el, hogy a **matched_pairs** változót elosztjuk a **total_pairs** változóval. Ezt követően a kapott eredményt terminálba kiíratjuk.

A következő kódrészletben a hasonlóságot fogjuk kiszámítani a V_{gc}^i és V_{gr} változók között.

```

1  mig = -1
2  intersection_indices = np.intersect1d(np.where(Vgr_i == 1)[0],
3  np.where(num_ones_per_block_i > 0)[0])
4  if len(intersection_indices) > 0:
5  # hasonlóság számítása az egyenlet alapján
6  numerator = np.sum(Vgc_i[intersection_indices] *
7  Vgr_i[intersection_indices])
8  denominator = np.sum(Vgc_i[intersection_indices] *
9  Vgr_i[intersection_indices] +
10 (1 - Vgc_i[intersection_indices]) *
11 Vgr_i[intersection_indices] +
12 Vgc_i[intersection_indices] * (1 -
13 Vgr_i[intersection_indices]))
14 mig = numerator / denominator
15 print("Hasonlóság  $V_{gc}^i$  és  $V_{gr}$  között:", mig)

```

Ehhez szükségünk lesz egy m_g^i nevű változóra, aminek alapértelmezett értékét mínusz egyre állítjuk. Ennek a változónak a segítségével fogjuk a futamot tartalmazó párokat tárolni. Ezután meg kell határoznunk az Ω_i és Ω_{gr} metszetét. Az **intersection_indices** változóban megkeressük azokat az indexeket, ahol mindkét vektorban egyes érték található. Ezek az indexek a két vektor közös elemei, amelyekre a hasonlóság kiszámítása történik. Ha létezik közös elem ($len(intersection_indices) > 0$), akkor kiszámítjuk a hasonlóságot az alábbi képlettel:

$$m_g^i = \frac{\sum(V_{gci} \cdot V_{gri})}{\sum(V_{gci} \cdot V_{gri}) + (1 - V_{gci}) \cdot V_{gri} + V_{gci} \cdot (1 - V_{gri})}$$

3.1. ábra: Hasonlóság kiszámítása V_{gc}^i és V_{gr} között [4]

A számlálóban azokat a helyeket számoljuk össze, ahol mindkét vektorban egyes érték szerepel. A nevezőben pedig az eseteket vesszük figyelembe, ahol legalább egyik vektorban 1-es van. Ez a hányados adja meg a két vektor közötti hasonlóság mértékét. Ha nincs

közös elem, akkor a m_i^g változó értéke mínusz egy marad, ami indikálja, hogy nincs észlelt futamba foglalt pár (gap-involved pair).

3.3 Csúcs alapú hasonlóság mérés

A következő részben csúcs alapú hasonlósági méréssel fogunk foglalkozni.

```

1 Vpr = np.zeros((len(blocks_original), 256), dtype=int)
2 tg = 0.5
3 NR = [i for i, mgi in enumerate(Ng_per_block) if mgi > tg]
4 for n in NR:
5     for k in range(256):
6         if np.any(blocks_original[n] > 0):
7             Vpr[n, k] = np.sum(blocks_original[n][blocks_original[n] >
8 0])
9 # EDR kiszámítása a Vpr alapján
10 Omega_pr = np.array([])
11 for n in NR:
12     if n < len(coexisting_gap_positions_all_blocks):
13         Omega_pr = np.union1d(Omega_pr,
14 coexisting_gap_positions_all_blocks[n])

```

Először létrehozunk egy referencia csúcs pozícióvektort, amit V_{pr} -nek fogunk hívni. Kezdetben egy olyan tömbként inicializáljuk, amelynek minden blokkjához tartozik egy sor, minden intenzitásértékhez egy oszlop, és minden eleme nullával van inicializálva. Ezt követően bevezetünk egy NR nevű változót, ami egy olyan lista, amelyben azoknak a blokkoknak az indexei találhatók meg, ahol a blokkban lévő futamok száma (mgi) meghaladja a tg küszöbértéket. A küszöbérték a [4] -es forrásból származik. Ezután két egymásba ágyazott for cikluson iterálunk végig. Az első ciklus az NR blokkjainak indexein megy végig n változóval, míg a második ciklusban k változóval minden ilyen referencia blokkhoz tartozó V_{pr} sorát inicializálja. Ha az adott blokkban található olyan pixel, aminek intenzitása nagyobb, mint nulla, akkor az if feltételünk teljesülni fog. Ilyenkor, a $Vpr[n, k]$ -ik eleme egyenlő lesz a $blocks_original$ tömb n -edik sorában lévő nem nulla értékek összegével.

Ezek után kiszámítjuk az EDR-t V_{pr} változó alapján. Ehhez létrehozunk egy Ω_{pr} változót, ami egy üres Numpy tömb kezdetben. Egy for ciklussal és egy n változóval végig iterálunk az NR -en, és megnézzük, hogy az n változó kisebb-e, mint a $coexisting_gap_positions_all_blocks$ lista hossza. Ha igen, akkor a $coexisting_gap_positions_all_blocks$ lista n -edik eleméhez tartozó futam pozíciókat hozzáadjuk az Ω_{pr} tömbhöz úgy, hogy minden elem csak egyszer szerepeljen. Az ismétlés elkerüléséhez az $np.union1d()$ függvényt használjuk, ami két tömb unióját hozza létre, jelen esetben Ω_{pr} és $coexisting_gap_positions_all_blocks[n]$, amelybe olyan elemek kerülnek, amik legalább az egyik tömbben szerepelnek.

Most a hasonlóságot fogjuk kiszámítani a V_{pc}^i és V_{pr} változók között.

```

1 mip = np.zeros(len(blocks_original)) # peak-involved pair értékek
2 for i in range(len(blocks_original)):
3     if i in NR:
4         intersection_indices = np.intersect1d(np.where(Vpr[i] > 0)[0],
5 np.where(blocks_original[i] > 0)[0])
6         if len(intersection_indices) > 0:
7             mip[i] = np.sum(blocks_original[i][blocks_original[i] > 0])
8 # ha nincs peak-involved pair, akkor -1
9 if not np.any(mip):
10     mip = -1
11 print("Peak-involved pair (mip):", mip)

```

Először létrehozunk egy m_p^i nevű tömböt, aminek hossza megegyezik a **blocks_original** tömbével és minden értékét nullával inicializáljuk. Ebben a változóban lesznek a peak-involved pair értékek. Ezután egy for ciklussal végigiterálunk a **blocks_original** mindegyik blokkján egy i változó segítségével. Ha i szerepel **NR** listában, akkor létrehozunk egy **intersection_indices** nevű változót. Ennek az értékadása úgy történik, hogy először megnézzük, melyek azok az indexek, ahol a $V_{pr}[i]$ és **blocks_original[i]** tömbök értékei nullánál nagyobbak. Ehhez az *np.where()* függvényt használjuk, ami visszaadja azokat az indexeket, ahol a feltétel teljesül. A függvények végén szereplő [0] rész azért van ott, mert a függvény visszaadhat több dimenziós tömböket is, mi viszont csak az elsőre vagyunk kíváncsiak, ezért ezt is odarakjuk feltételnek. Az *np.intersect1d()* függvény azonosítja azokat az elemeket, amelyek mindkét tömbben megtalálhatóak, azaz megkeresi a két tömb metszetét. Ezután megnézzük, hogy az **intersection_indices** hossza nagyobb-e nullánál, vagyis szerepel-e benne elem. Ha igen, akkor az **intersection_indices** tömb i -edik indexénél összegzi *np.sum()* függvény segítségével azokat az elemeket, amik nagyobbak nullánál. A kifejezésben az első **blocks_original[i]** segítségével a blokkot választjuk ki, a másodikkal pedig a pozitív értékeket szűrjük ki. Ha ezek után a műveletek után se találunk **peak-involved pair**-t, akkor m_p^i változó értékét mínusz egyre állítjuk. Az előbb leírtakat úgy tudjuk kód formájában megvalósítani, hogy *np.any()* függvényt használunk az m_p^i változóra. Ha bármelyik elem értéke nulla, a függvény **true** értékkel fog visszatérni és kiírjuk az m_p^i tömböt. Viszont, ha minden elem nulla, akkor a függvény **false** értékkel fog visszatérni és az m_p^i értékét -1-re állítjuk, majd ezt az értéket kiírjuk.

4. SVM

4.1. SVM jellemzők előállítása

A következő kódrészlet az **SVM** (Support Vector Machine) algoritmusnál a jellemzők előállítását mutatja be.

```

1  # Üres DataFrame létrehozása
2  df = pd.DataFrame(columns=['label', 'mp', 'mg'])
3
4  # képek mappájának elérési útvonala
5  image_folder = 'UCID Database/UCID Database/UCID1338'
6
7  # lista a képek elérési útjainak tárolására
8  image_paths = [os.path.join(image_folder, filename) for filename in
9  os.listdir(image_folder)
10                 if filename.endswith('.tif')]
11 # első 10 képet használjuk
12 image_paths = image_paths[:10]
13 # Üres változó a kombinált képek tárolására
14 combined_images = []
15 #mig értékek tárolására
16 mig_values = []
17 #mip értékek tárolására
18 mip_values = []
19 # labelek tárolására
20 labels = []

```

Először is, kelleni fog egy **Dataframe**, amiben az adatokat fogjuk tárolni és kezelni, ezt egy **df** nevű változóban fogjuk eltárolni. Ez egy táblázatszerű adatszerkezet, ami több oszlopból áll és minden oszlopban különböző típusú adatot fogunk tárolni. A mi esetünkben három oszlop lesz, az elsőben fognak szerepelni az **mⁱ_p** értékei (ez a változó tárolja a csúcsot tartalmazó párokat), ezt követi az **mⁱ_g** oszlopa (futamokat tartalmazó párok tárolására való) és végül az utolsó oszlop a **labels**-é lesz, amiben a blokkokhoz tartozó címkéket fogjuk tárolni. Ezek nullás vagy egyes értéket vehetnek fel elhelyezkedésüktől függően. Az adatszerkezetnek megadjuk az oszlopait és majd csak később fogjuk feltölteni adatokkal. Ahhoz, hogy fel is töltsük adatokkal, kelleni fog nekünk jó pár kép. A kódrészlet következő soraiban az **image_folder** változóban megadjuk az elérési útvonalat a képek mappájához. Az **image_paths** változóban az **os.listdir()** függvény segítségével kilistázzuk az összes fájlt az **image_folder** változóban lévő mappából, ezt követően egy **if** feltétellel kiszűrjük a **.tif** kiterjesztésű fájlokat. Az **os.path.join()** függvénnyel összeillesztjük a mappa elérési útvonalát az **image_folder** változóból és a fájl neveket a korábbi szűrés eredményéből, így most már tudni fogjuk a fájloknak a teljes elérési útját. Ezután az **image_paths** lista első 10 képét kiválasztjuk, hogy fel tudjuk tölteni a **df** változó oszlopait adatokkal. Természetesen a 10 helyére más értéket is lehetne írni, hogy minél nagyobb teszthalmazunk legyen. Ezt követően pedig létrehozunk négy változót, rendre az **mⁱ_g**, **mⁱ_p**, **labels** és a **combined_img** változók tárolásához, **mig_values**, **mip_values**, **labels** és **combined_images** néven. Az utolsó változó egy olyan tömb, amibe olyan képeket fogunk belerakni, amiknél egy eredeti képet (ciklusban éppen adott kép) és egy CLAHE által módosított képet fogunk összeadni, hogy egyesített képeket kapjunk.

```

1  for img_path in image_paths:
2      # Kép beolvasása szürkeárnyalatosan
3      img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
4      # Kép méretének meghatározása
5      N, M = img.shape
6      # Maszk létrehozása
7      mask = np.zeros((N, M), dtype=np.uint8)
8      center_N = N // 2
9      center_M = M // 2
10     mask[center_N - 64:center_N + 64,
11          center_M - 64:center_M + 64] = 1

```

A következő kódrészletben belépünk egy for ciklusba, amiben a listázott képeken iterálunk végig. Ebbe a ciklusba még sokáig benne is fogunk maradni, csak az érthetőség kedvéért több kisebb részre lett felosztva. A cikluson belül mindig első lépésként beolvassuk az aktuális képet szürkeárnyaltos módban. Ezt követően meghatározzuk az adott kép magasságát és szélességét, hogy ki tudjuk számolni a maszkok méretét. Azért fontos minden iterációban lekérni a kép méreteit, mivel egyik kapott képünk lehet mondjuk egy álló kép, míg a következő egy fekvő tájolású. A **mask** változó létrehozásakor először azt feltöltjük egy nullmátrixszal, aminek a mérete megegyezik a beolvasott képével. Ezután meghatározzuk a maszk közepét, ahol egy 128x128-as területen egyesre fogjuk állítani a maszk értékét, többi helyen marad nulla. Ezt azért csináljuk meg, mert ezen a kijelölt területen fogunk kontrasztjavítást végezni a kapott képeken, így tudni fogjuk pontosan hol lettek módosítva a képek és az eredményünket a legvégén tudjuk ellenőrizni is, hogy tényleg megfelelő értékeket kaptunk-e.

```

1  # CLAHE maszk és eredeti maszk létrehozása
2  clahe_mask = mask
3  orig_mask = 1 - mask
4  # CLAHE algoritmus létrehozása és alkalmazása
5  clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
6  img_clahe = clahe.apply(img)
7  # Maszkok alkalmazása a képekre
8  img_orig_masked = img * orig_mask
9  img_clahe_masked = img_clahe * clahe_mask
10 # Eredeti és CLAHE maszkolt képek összeadása
11 combined_img = img_orig_masked + img_clahe_masked
12 # Hozzáadás a kombinált képek listához
13 combined_images.append(combined_img)

```

Az alábbi kódrészletben először is létrehozuk a CLAHE maszkot, amit egyszerűen lemásolunk a korábban kiszámított **mask** változóról és a kapott eredményt a **clahe_mask** változóban tároljuk el. Az eredeti maszkot pedig úgy kapjuk meg, hogy egyből kivonjuk a **mask** változó összes elemét (azaz vesszük a **mask** változó komplementerét), ezt pedig az **orig_mask** változóban tároljuk el. Ezután egy CLAHE objektumot hozunk létre a **cv2.createCLAHE()**

függvény segítségével, ahol a *clipLimit* paraméter határozza meg a kontraszt korlátozását, a *tileGridSize* pedig az adaptív kiegyenlítési blokkok méretét határozza meg. A kapott objektumot a **clahe** változóban tároljuk el. Az **img_clahe** változóban pedig vesszük az adott iterációban lévő képet és alkalmazzuk rá a **clahe.apply()** függvény segítségével az eljárást. Most, hogy megvan az eredeti képünk és a CLAHE eljárással módosított képünk, tudjuk alkalmazni a korábban kapott maszkokat a képekre. A maszkolt képeket az **img_orig_masked** és **img_clahe_masked** változókban fogjuk eltárolni, az elsőnél a beolvasott képet szorozzuk meg az **orig_mask** változóval, azaz a **mask** változó komplementerével, az utóbbinál pedig a CLAHE által javított képet szorozzuk meg a **clahe_mask** változóval. Az újonnan kapott **img_orig_masked** és **img_clahe_masked** változókat most összeadjuk. Így az eredeti képen azok a területek, ahol a maszk értékei egyesek, változatlanok maradnak, míg azokon a területeken, ahol a maszk értékei nullák, a CLAHE által javított kép értékei fognak megjelenni, ezzel létrehozva egy kombinált képet. Ezt a képet a **combined_img** változóban tároljuk el és végül hozzáadjuk ezt a **combined_images** tömbhöz, amiben az összes ilyen képet fogjuk tárolni.

Az alábbi részben láthatjuk a címkék hozzáadását a hozzájuk tartozó **labels** tömbhöz.

```

1  Nb = 32
2  blocks_combined = [combined_img[y:y + Nb, x:x + Nb] for y in range(0, N,
3  Nb) for x in range(0, M, Nb)]
4  clahe_blocks = [clahe_mask[y:y + Nb, x:x + Nb] for y in range(0, N, Nb)
5  for x in range(0, M, Nb)]
6
7  # Labels létrehozása blokkonként a CLAHE maszk alapján
8  block_labels = [] # blokkok címkéinek tárolása
9  for block, mask_block in zip(blocks_combined, clahe_blocks):
10     if np.any(mask_block == 1):
11         block_labels.append(1)
12     else:
13         block_labels.append(0)
14 labels.append(block_labels)

```

Első lépésként az *Nb* változóban megadjuk, hogy mekkora méretű blokkokra szeretnénk felosztani az adott képet. Jelen esetben 32x32 pixeles blokkokat kapunk. Ezt követően az iterációban lévő **combined_img** és **clahe_mask** változókat is felosztjuk blokkokra. A kapott eredményeket eltároljuk a **blocks_combined** és **clahe_blocks** változókban. Az első változó segítségével fogjuk később az **m_g** és **m_p** értékeit meghatározni, míg a másikkal az adott blokkhoz tartozó címkéket. Ezek után létrehozunk egy **block_labels** változót, amit kezdetben üres tömbként inicializálunk. Ebbe fogjuk tárolni a kép blokkjaihoz tartozó címkéket, amiket mindig hozzáadunk a **labels** tömbhöz. Az inicializálást követi egy for ciklus, amiben a **block** és **mask_block** változó segítségével végig iterálunk a **blocks_combined** és **clahe_blocks**

listákon párhuzamosan. Erre azért van szükség, hogy egy képi blokkhoz címkét tudjunk rendelni annak megfelelően, hogy a `clahe_mask` blokk tartalmaz-e egyes értéket. Ezek alapján adunk hozzá nullás vagy egyes címkét a **block_labels** változóhoz.

A következő szakaszban meg kell keresnünk a ciklus adott képének az m_g^i és m_p^i értékeit. Ezt viszont már a harmadik fejezetben elvégeztük. Mivel nagyon sok változó függ egymástól, ezért szinte minden értéket újra ki kell számolni és emlékeztetésül még mindig a három kódrészlettel ezelőtti for cikluson belül vagyunk. Miután átmásoljuk az SVM rész előtti kódot a for cikluson belülre, pár helyen át kell neveznünk a **block_original** változót **blocks_combined** változóra, hogy ne az eredeti képen történjen mindig a számítás, hanem a cikluson belül lévő adott képen. Továbbá még ezt a két sort hozzá kell adnunk azokhoz a részekhez, ahol megtörtént az m_g^i és m_p^i változók kiszámítása, hogy elmentsük az értékeket tömbökbe és később a Dataframe feltöltésénél tudjuk használni a változókat.

```
1 mig_values.append(mig)
2
3 mip_values.append(mip)
```

Most, hogy megvan minden változónk, elkezdhetjük feltölteni a Dataframe-et adatokkal.

4.2. Osztályozás nem-lineáris SVM-mel

Az alábbi kódrészletben adatokat fogunk gyűjteni a pandas Dataframe-ben.

```
1 df = pd.DataFrame()
2 for i in blocks_original:
3     df['label'] = ... # 0 vagy 1 attól függően, hogy valódi vagy
4     # manipulált a blokk
5     df['mp'] = mip[i]
6     df['mg'] = mig[i]
```

A Dataframe egy táblázatszerű struktúra a pandas könyvtárban. Oszlopokból áll és különböző típusú adatokat tárolhatunk benne. Első lépésként létrehozunk egy üres dataframe-et a `pd.DataFrame()` függvénnyel, eredményünket a **df** változóban tároljuk el. A következő lépésben egy for ciklussal végig iterálunk a **blocks_original** tömbön egy **i** változó segítségével. A cikluson belül létrehozunk három új oszlopot, az elsőt **label** néven, később majd nulla, illetve egyes értékek fognak ide kerülni, attól függően, hogy a blokk valós vagy manipulált. A második oszlop az **mp** nevet fogja kapni, ez a peak-involved pair (m_p^i [i]) értékeit fogja tárolni az adott blokkra vonatkozóan. Az utolsó oszlop pedig az **mg** nevet viseli, itt a futamba foglalt pár (m_g^i [i]) értékeit tartalmazza az adott blokkra nézve.

A következő részletben a tanító- és tesztalmaidatokat fogjuk szétválogatni.

```
1 X_train, X_test, Y_train, Y_test = train_test_split(X_all, Y_all,
2 test_size=0.2, random_state=42)
```

A `train_test_split()` függvény segítségével az adathalmazt két részre osztjuk. Lesz egy tanítóhalmazunk az **X_train**, **Y_train** változókkal, és egy teszhalmazunk **X_test**, **Y_test** változókkal. Paramétereket tekintve a **test_size** indikálja, hogy az adatok hány százalékát szeretnénk teszhalmazként használni, ami jelen esetben 20% lesz. A **random_state** paraméterrel az adatokat véletlenszerűen osztjuk szét, ezzel biztosítjuk, hogy ugyanaz a randomizálás lesz használva akármikor futtatjuk a kódot. Továbbá biztosítjuk, hogy az adatok ugyanolyan arányban lesznek elosztva.

Ezek után következik az SVM betanítása.

```
1 param_grid_svm = {'C': [1, 10, 100],
2                  'gamma': [0.01, 0.001, 0.0001],
3                  'kernel': ['rbf']}
4 grid = GridSearchCV(SVC(), param_grid_svm, refit=True, verbose=3)
5 grid.fit(X_train, Y_train)
```

Először definiálunk egy paraméter tartományt. Ehhez létrehozunk egy szótárt **param_grid_svm** néven, ami tanítás során kipróbálandó paramétereket tárolja. Ilyen például a **c** (regularizációs paraméter), **gamma** (RBF kernel együtthatója) és a **kernel** (kernel típusa). Az alábbi sorban használatba kerül a **GridSearchCV**, ami egy osztálya a scikit-learn könyvtárnak, amit hálózatos validációhoz használunk. A mi esetünkben az SVM paramétereiben végez keresést és kereszt-ellenőrzést végez az adathalmazon. Paramétereket nézve az `SVC()` példányosít egy Support Vector Classifiert. Erre azért van szükség, mert a **GridSearchCV** osztálynak kell egy becslő osztály, amelynek paramétereit optimalizálni akarja a rákövetkező **param_grid_svm** paramétertérben. A **param_grid_svm** tartalmazza a korábban említett **c**, **gamma** és **kernel** paramétereket. A `refit=True` azt jelzi, hogy a legjobb paraméterek megkeresése után az osztály újra illeszti az optimális paraméterekkel a modellt az összes tanító adatra. A `verbose=3` beállítás jelzi, hogy részletes kimenetet kapunk a keresés folyamatáról. Minél nagyobb számot írunk ide, annál több információt kapunk a keresésről. Végül a rácskeresési modellt ráillesztjük a tanító adatokra, hogy megtaláljuk a legjobb paramétereket a paramétertartományon belül. Ehhez a `fit()` függvényt használjuk, az **X_train** és **Y_train** a tanító adatok. **X_train** az adatok jellemzőit, **Y_train** pedig a hozzájuk tartozó címkéket tartalmazza.

Az utolsó kódrészletben a modell értékelését fogjuk elvégezni. Ebben a részben először a mérőszámok szerinti kiértékelést fogjuk elvégezni, majd az eredmények kiíratását.

```

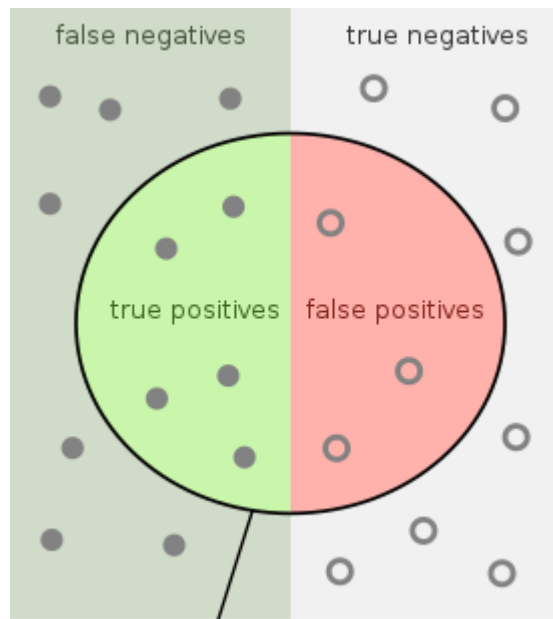
1  y_pred = grid.predict(X_test)
2  dump(grid, 'gridsearch_svm2.joblib') # csak elmentjük a modellt
3  prec_score = precision_score(Y_test, y_pred)
4  accu_score = accuracy_score(Y_test, y_pred)
5  rec_score = recall_score(Y_test, y_pred)
6  f1 = f1_score(Y_test, y_pred)
7  conf_matrix = confusion_matrix(Y_test, y_pred)
8  print('precision_score: ', prec_score)
9  print('accuracy_score: ', accu_score)
10 print('f1_score: ', f1)
11 print('recall_score: ', rec_score)
12 print('confusion_matrix: ', conf_matrix)
13 conf_disp = ConfusionMatrixDisplay(conf_matrix,
14 display_labels=["helyes", "teves"])
15 conf_disp.plot()
16 plt.show()

```

Először is az **X_test** adatok alapján meghatározzuk a modell predikcióit a *grid.predict()* függvény segítségével, majd a kapott modellt elmentjük a **gridsearch_svm2.joblib** fájlba a *dump()* függvény alkalmazásával. Most következik a mérőszámok szerinti kiértékelés. A **precision_score** egy metrika, ami megmutatja, hogy az összes pozitív előrejelzés közül hány százalék valósult meg valóban pozitívan. Általában hamis pozitív arány csökkentésére használják.

Ahhoz, hogy megértsük ez miért hasznos számunkra, vezessünk be pár új fogalmat az osztályozási modellek értékelésével kapcsolatban. Az első ilyen fogalom a **true positive**, vagy másnéven **tényleges pozitív** lesz. Ebben az esetben a modell helyesen azonosítja a pozitív esetet, tehát ha mondjuk teszteljük ezt az SVM modellt, és a modell azt állítja, hogy a kapott kép manipulált, és valóban az, akkor ezt az esetet **tényleges pozitívnak** tekintjük. Lehetnek olyan esetek is, amikor a modell tévesen azonosítja a pozitív esetet. Például a modell azt mondja, hogy egy kép manipulált, de igazából nem az, akkor ezt az esetet **false positive-nak**, vagy **hamis pozitívnak** nevezzük. A következő eset akkor történik meg, amikor a modell helyesen azonosítja a negatív esetet. Ilyenkor a modell egy nem manipulált képről azt állítja, hogy nem volt manipulálva. Ezt az esetet **true negative-nak** vagy **tényleges negatívnak** nevezzük. Az utolsó esetről pedig a modell tévesen azonosít egy negatív esetet. Ez akkor történik meg, amikor a modell egy manipulált képre azt állapítja meg, hogy nem manipulált képet kapott. Ezt az esetet pedig **false negative-nak** vagy **hamis negatívnak** nevezzük. Számunkra persze az lenne az optimális, hogyha minél nagyobb lenne a **tényleges pozitív** esetek száma, miközben a hamis pozitívok és hamis negatívok számát a lehető legalacsonyabbra

csökkentjük. Az elmondottakat könnyebben is tudjuk értelmezni, ha a 4.1 ábrát megtekintjük. Most, hogy ezekkel a fogalmakkal tisztában vagyunk, haladhatunk tovább a többi változóval.

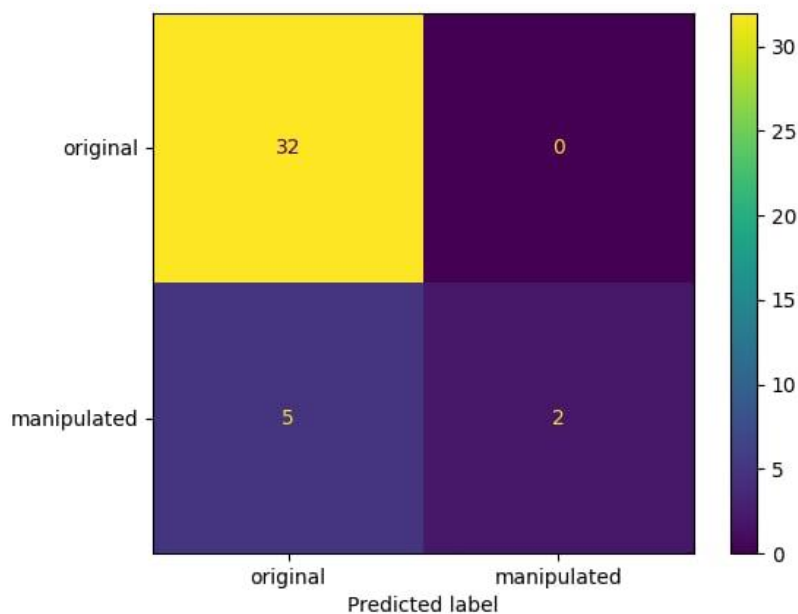


4.1. ábra: Modellek kiértékelése [7]

Az **accuracy_score** metrika az összes helyesen osztályozott minta arányát mutatja az összes mintához képest. Ez az egyik leggyakrabban használt metrika modell teljesítmények értékelésére. A **recall_score** metrika a helyesen pozitív esetek arányát mutatja a pozitívnak becsült esetek között. Hamis negatív arány csökkentése érdekében szokták használni. Az **f1_score** egy súlyozott átlagot határoz meg az **accuracy_score** és **recall_score** között és azt jelzi, hogy mennyire jó az egyensúly a két változó között. A **confusion_matrix** pedig egy olyan táblázat, ami összehasonlítja az eredeti és becsült osztályokat. A sorok az eredeti osztályokat, az oszlopok pedig a becsült osztályokat tartalmazzák. A táblázat azt mutatja, hogy a modell melyik osztályba sorolta helyesen vagy tévesen a tesztelési adatokat. Ezt követően kiíratjuk az összes új változónkat, hogy lássuk az eredményeket. Az utolsó változó, amit létrehozunk a **conf_disp** nevet fogja viselni. A változóban egy igazságmátrixot fogunk tárolni, amit a *ConfusionMatrixDisplay()* függvény segítségével hozunk létre. A függvény két fő paramétere a *conf_matrix* (táblázat, amit meg akarunk jeleníteni) és a *display_labels*, amiben egy listát kell megadnunk, amik az igazságtábla oszlopainak és sorainak címkéi lesznek. Miután végeztünk az igazságmátrix létrehozásával, meghívjuk a *plot()* függvényt, hogy kirajzoljuk a kapott mátrixot. Végül a *plt.show()* függvénnyel megjelenítjük a diagramot a képernyőn.

5. EREDMÉNYEK

A program futtatása után az 5.1 ábrán látható igazságmátrix fog felugrani a program ablakának jobb oldalán található Plots fülben, ami a Notifications és Database fül alatt található. Az oszlopok a modell által előre jelzett címkéket mutatják, míg a sorok a valós címkéket jelölik. Az ábra első oszlopáról leolvashatjuk, hogy 32 esetben helyesen azonosította a modell a címkét (**tényleges pozitív**), 5 esetben pedig tévesen azonosított eredetinek egy címkét (**hamis pozitív**). A második oszlopban pedig azt figyelhetjük meg, hogy egyetlen esetben sem azonosított tévesen címkét (**hamis pozitív**), és 2 esetben pedig helyesen talált meg manipulált címkéket (**tényleges negatív**).



5.1 ábra: futtatás után kapott eredmény

Ezen kívül viszont a terminálban is kaptunk vissza pár érdekes információt, amit az 5.2 ábrán figyelhetünk meg.

```
precision_score: 1.0
accuracy_score: 0.8717948717948718
f1_score: 0.4444444444444444
recall_score: 0.2857142857142857
```

5.2 ábra: terminálban kapott eredmények

Az első érték a **precizitás**, ami azt méri, hogy a modell által manipuláltnak jelölt adatok közül mennyi volt valójában manipulált. Az egyes érték azt jelenti, hogy a modell nem azonosított tévesen blokkokat, viszont az 5.1 ábrán láthatjuk, hogy 5 olyan eset is volt, amikor manipuláltnak hitt egy eredeti képet, tehát ezen a számításon még javítani kell. A következő

érték a **pontosság**, itt az összes helyes előrejelzés arányát nézzük meg az összes előrejelzéshez viszonyítva. Eredményül 87.18%-os helyes előrejelzést kaptunk, ami egy egész jó aránynak mondható. A következő változó az **f1-score**, ez a **precizitás** és az **érzékenység** harmonikus átlaga. Ennek a segítségével egy jó mutatót kaphatunk a modell általános teljesítményéről, főleg abban az esetben, amikor az osztályok kiegyensúlyozatlanok. A kapott értékünk arra utal, hogy a precizitásunk magas, az **érzékenységünk** viszont alacsony és a modell nem talált meg sok manipulált esetet. Az utolsó eredményünk pedig az érzékenység, ez a változó méri, hogy a valós manipulált esetek között hányat talált meg a modell. A kapott értékből azt deríthetjük ki, hogy ezeknek 28,57%-át találta meg, ami elég alacsony.

6. A PROGRAM HASZNÁLATA

A program futtatásához a JetBrains termékcsaládból a Pycharm programra lesz szükségünk. A feladat a Python 3.11.2-es verziójával lett implementálva. A zip fájl kicsomagolása után láthatjuk, hogy a mappában szerepel egy kontrasztjavított kép `input_image.png` néven, egy `kontrasztjav.py` fájl, amiben az algoritmus lett implementálva és egy UCID Database nevezetű mappa, amiben képek szerepelnek az SVM jellemzők előállításához. A kicsomagolt mappát megnyithatjuk egy új projektben a Pycharm programban és a Python fájl kiválasztása után már tudjuk is futtatni a programot vagy az ablak tetején lévő zöld nyíl gomb megnyomásával vagy a `shift+F10` billentyűkombinációval. A program futtatása után a terminálban megjelenik minden részeredmény, amelyekről az implementáció leírása során szó esett. A program jobb oldali sávjában pedig a Plots fülben meg fog jelenni az igazságtábla, amiben láthatjuk a **tényleges pozitívok**, **tényleges negatívok**, **hamis pozitívok** és **hamis negatívok** eloszlását.

Irodalomjegyzék

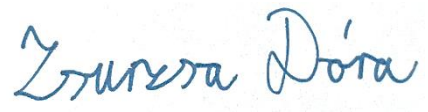
- [1] Lincoln-Calhoun kompozit kép forrása
<https://iconicphotos.wordpress.com/2010/04/24/lincoln-calhoun-composite/>
- [2] Katalin hercegné Anyák napi posztja és magyarázata
 - a. <https://x.com/KensingtonRoyal/status/1766750995445387393>
 - b. <https://x.com/KensingtonRoyal/status/1767135566645092616>
- [3] Tampered Region Detection on Digital Images by Efficient SVM Classifier
 - a. <https://www.jetir.org/papers/JETIR1510007.pdf>
- [4] Gang Cao, Yao Zhao, Senior Member, IEEE, Rongrong Ni, Member, IEEE, and Xuelong Li, Fellow, IEEE: Contrast Enhancement-Based Forensics in Digital Images
 - a. https://www.researchgate.net/publication/260523455_Contrast_Enhancement-Based_Forensics_in_Digital_Images
- [5] Emberek fotózása és a jog
 - a. https://www.fotozz.hu/cikket_megmutat?cikk_ID=68.
 - b. Polgári törvénykönyv: [2013. évi V. törvény - Nemzeti Jogszabálytár - njt.hu](https://www.szerzo.hu/torveny/2013-05-01/V-torveny)
- [6] Exposing Image Forgery through Contrast Enhancement: Priya Ann Koshy, Kavitha N. Nair
 - a. <https://www.ijsr.net/getabstract.php?paperid=NOV152185>
- [7] Modellek kiértékelése (osztályozás és rangsorolás)
 - a. https://en.wikipedia.org/wiki/Sensitivity_and_specificity

Nyilatkozat

Alulírott Zsurzsa Dóra, Programtervező informatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, Programtervező informatikus BSc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Diplomamunka Repozitóriumban tárolja.



2024.05.22.

Dátum

Aláírás

Mellékletek

1. kontrasztjav_lok.py
2. input_image.png
3. UCID Database mappa
4. gridsearch_svm2.joblib