1. Introducere
   1. Descrierea aplicatiei
      1. Scopul aplicației
      2. Descrierea NDVI
      3. Componenta de clasificare
      4. Baza de date hadoop
      5. Componenta spring și JavaScript
2. Descrierea componentelor
   1. Componenta de clasificare
      1. Librării importante folosite
      2. Modeul de functionare
   2. Baza de date Hadoop
      1. Principii si componente hadoop folosite
      2. Masina virtuala Cloud Era Hortonworks
      3. Structura datelor
      4. Enpoint-uri folosite
   3. Componenta Spring
   4. Componenta Angular
3. Concluzii
4. **Bibliografie**

1 Introducere

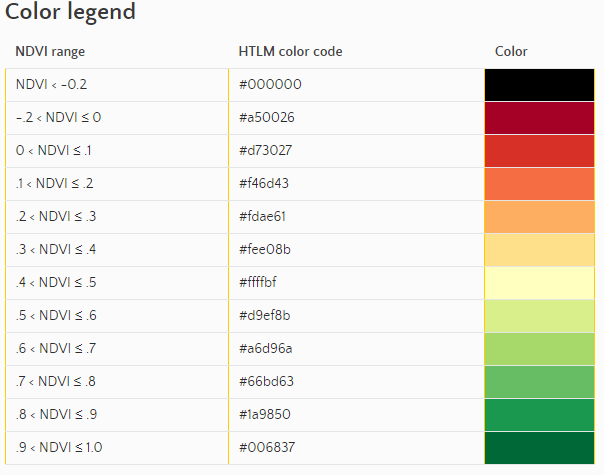
1.1 Descrierea aplicației

1.1.1 Scopul aplicatiei

Scopul aplicației este de a analiza parcele de pamant pentru a facilita activiatea fermierilor si pentru a genera statistici la nivel de proprietate, localitate, tara etc. Proiectul se bazază pe poze satelitare ce sunt analizate cu ajutorul componentei python de clasificare. Deoarece componenta python este relativ lenta, din cauza algoritmilor pe care ii ruleaza este nevoie de stocarea datelor pentru un acces mai rapid. De exemplu, pentru o parcelă de pământ de 0.5 km^2, algoritmul o imparte in 70 de parti diferite, fiecare cu o multitudine de puncte, categorizate in funcție de indicele NDVI. Din motive de scalabilitate este nevoie de implementare unei baze de date NOSql distribuita. Am ales să folosesc Hadoop ca tehnologie pentru baza de date.

1.1.2 Descrierea NDVI

NDVI (normalized difference vegetation index) este un indicator grafic care este folosit pentru a cuantifica vegetația verde din imagini. Indicatorul poate lua valori intre –1 și 1. Valorile ce se apropie de –1 corespund corpurilor de apă. Valorile din vecinătatea și interiorul intervalului (-1, 1) corespund solurilor neroditoare: nisip, zapadă sau stânci. Valorile pozitive mici, din zona 0.2-0.4 reprezintă zone acoperite de arbuști și pajiști. Valorile ce se apropie de 1 corespund pădurilor din zonele temperate și tropicale.



(sursă <https://custom-scripts.sentinel-hub.com/sentinel-2/ndvi/>#)

**2.1 Componenta de clasificare**

Componenta de clasificare este un serviciu scris in python ce expune 2 endpoint-uri. Acestea returneaza ca raspuns un json, respectiv un json linked list ce contine date referitoare la clasificarea parcelei in 3 clase, in functie de indicele NDVI.

* [http://{ip}:{port}/ndvi/api/json/v1/ndvi-classification](http://127.0.0.1:5000/ndvi/api/json/v1/ndvi-classification)
* http://{ip}:{port}/ndvi/api/jsonld/v1/ndvi-classification

În cazul de față serviciul ruleaza pe localhost, pe portul 5000.

**2.1.1 Librarii importante folosite**

**Flask** este un micro-framework web scris în Python. Este clasificat ca microframework deoarece nu necesită instalarea altor librării sau tool-uri. De asemena am folosit ca extensie peste Flask, flask\_restx, datorită suport-ului pentru Swagger. Cele 2 endpoint-uri sunt documentate cu Swagger la [http://{ip}:{port}/](http://127.0.0.1:5000/ndvi/api/json/v1/ndvi-classification) .

Cele 2 metode sunt expuse cu ajutorul adnotărilor puse la dispozitie de Flask:

flask\_app = Flask('NDVIClassification')

app = Api(app=flask\_app,

version='1.0',

title='NDVI classification',

description='Classify Images by the NDVI index')

name\_space = app.namespace('ndvi', description='NDVI-specific API s')

model = app.model('Input\_Json\_Model',

{

'polygonCoordinates': fields.String(

required=True,

description='Coordinates of the polygon to be analysed.',

help='Coordinates cannot be left unspecified.',

example=polygonCoordinatesString

),

'clientId': fields.String(

required=True,

description='Coordinates of the polygon to be analysed.',

help='Coordinates cannot be left unspecified.',

example=token

)

})

@name\_space.route("/api/json/v1/ndvi-classification")

class JsonApi(Resource):

@name\_space.route("/api/jsonld/v1/ndvi-classification")

class JsonldApi(Resource):

Cele 2 rute expun 2 metode de tip POST ce primesc in body-ul request-ului un JSON cu 2 atribute obligatorii:

* Primul atribut ‘polygonCoordinates’ conține coordonatele (latitudine și longitudine) punctelor care determină parcela de pământ ce va fi analizată. Un exemplu ar fi cel din Swagger: "27.199243,45.910026,27.209468,45.911885,27.209607,45.906525,27.200563,45.904793".
* Al doilea atribut ‘clientId’ este un token static folosit ca o formă simplă de autentificare. Id-ul este: "36aef768-3076-43ad-8566-4818fd82c73c"

Prin apelarea metodei run ce primește ca argumente ip-ul si portul vom porni serviciul de clasificare

flask\_app.run(host='0.0.0.0')

Portul standard este portul 5000.

**Pillow**

Pillow este o librărie folosită pentru a adăuga capabilități de procesare a imaginilor interpretorului Python. Am folosit doar modulul Image al librăriei ce conține o clasă cu același nume care este folosită pentru a reprezenta o imagine Pillow. Am folosit funcția open pentru a deschide imaginea și pentru a efectua operații asupra imaginii.

**OpenCV-Python**

OpenCV este o librărie de funcții de programare cu algoritmi de si machine learning computer vision. Librăria fost inițial dezvoltată de Intel. Librăria este independentă de platformă si gratis def olosit cu licență open-source Apache 2.

OpenCV are suport pentru mai multe limbaje precum C++, **Python** sau Java și poate fi folosită pe mai multe platforme precum Windows, Linux, Os X, Android, iOS etc.

De completat----

**2.1.2 Descrierea serviciului**

Pentru a explica modul in care funcționează componenta vom analiza procesul incepând în ordine de la primirea datelor inițiale pană la generarea și returnarea json-ului de output. Singura diferență între cele două tipuri de json-uri returnate de cele 2 endpoint-uri este modul în care acestea sunt construite. Pentru a explica modul de funcționare a aplicației se va face referire la endpoint-ul pentru json simplu: [http://{ip}:{port}/ndvi/api/json/v1/ndvi-classification](http://127.0.0.1:5000/ndvi/api/json/v1/ndvi-classification).

Primul pas este verificarea token-ului. Token-ul din request trebuie sa se potrivească cu cel setat în Properties.py. Apoi, prin apelarea funcției *getBBOXFromParcelCoordinates,* ce primește ca date de intrare coordonatele parcelei din request (*args['polygonCoordinates']*) se returnează bounding box-ul definit de aceste puncte.

Bounding Box (prescurtată si BBOX) este o zonă dreptunghiulară definită de 2 puncte (adica 2 perechi latitudine- longitudine). In cazul nostru avem nevoie de bounding box-ul ce contine toate puntele primite la input. Pentru a determina cele 2 puncte ale bounding box-ululi funcția extrage longitudinea maximă si minimă și latitudinea maximă si minimă:

------Poate exemplu de bounding box--------

def getBBOXFromParcelCoordinates(coordinatesList):

coordinatesBBOX = []

xList = coordinatesList[0::2]

yList = coordinatesList[1::2]

coordinatesBBOXMinX = min(xList)

coordinatesBBOXMinY = min(yList)

coordinatesBBOXMaxX = max(xList)

coordinatesBBOXMaxY = max(yList)

coordinatesBBOX.append(coordinatesBBOXMinX)

coordinatesBBOX.append(coordinatesBBOXMinY)

coordinatesBBOX.append(coordinatesBBOXMaxX)

coordinatesBBOX.append(coordinatesBBOXMaxY)

#print(coordinatesBBOX)

coordinatesBBOX = epsg4326ToEpsg3857(coordinatesBBOX)

return coordinatesBBOX

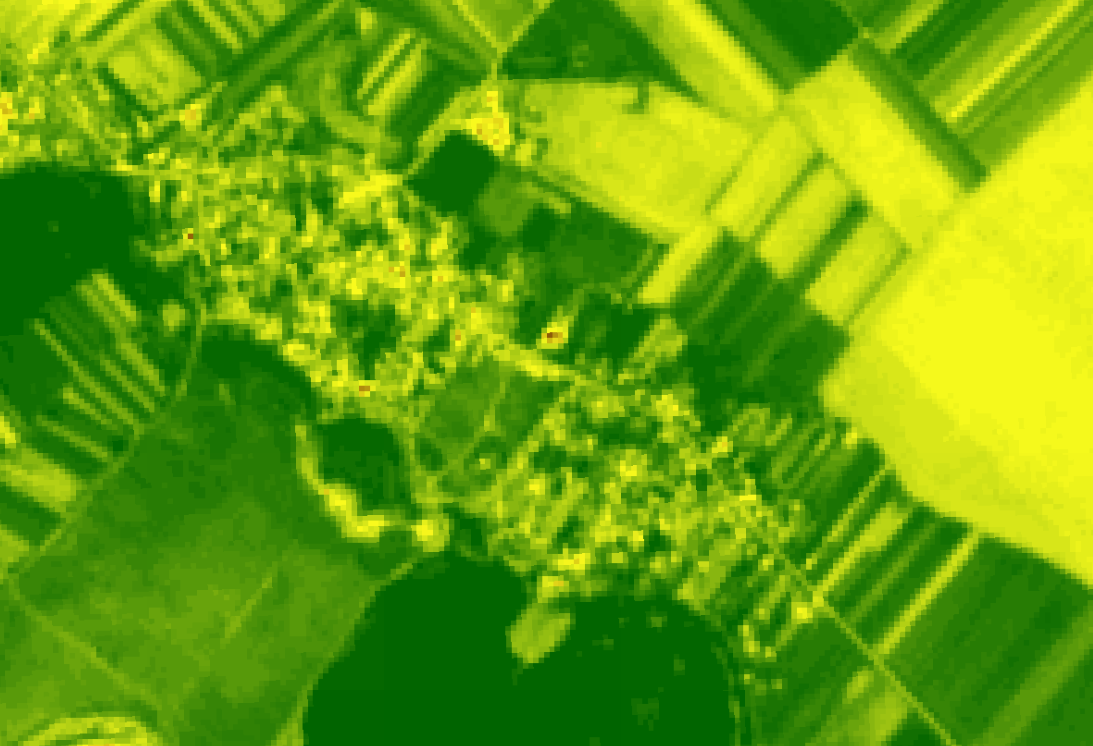
De asemenea, funcția apelează *epsg4326ToEpsg3857(coordinatesBBOX)* care trece coordonatele din input din sistemul epsg4326 în sistemul epsg3857. În proiect este definită si funcția inversă.

**Serviciile Terrascope**

Următorul pas este obținerea imaginii ce va fi analizată prin folosirea a două servicii puse la dispoziție de platforma terrascope.be.

Terrascope este o platformă belgiană ce oferă acces nerestrcționat pentru toți utilizatorii printre altele la date satelitare. Satelitul Sentinel-2 obține date optice pentru Europa și Africa centrală. Cu ajutorul datelor de la Sentinel 2 Terrascope aplică un filtru NDVI asupra imaginilor satelitar. Următorul exemplu este o imagine decupată folosind platforma terrascope de vizionare a flitrelor de vegetație <https://viewer.terrascope.be/> cu filtrul NDVI aplicat asupra satului Cucuieți, județul Teleorman si asupra zonelor limitrofe.

**Serviciul pentru imagini NDVI**



Unul dintre serviciile terrasope folosit este <https://services.terrascope.be/wms/v2> de tip GET. Cu ajutorul acestui endpoint se poate obține o imagine ca cea din exemplul anterior. Printre parametrii mai importanți se regăsesc:

* layers: stratul care va fi aplicat, in cazul de față: *CGS\_S2\_NDVI*
* format: în aplicație este folosit: *image/png*)
* srs: sistemul de coordonate folosit: *EPSG:3857*
* time: ziua în care a fost facută poza (terrascope stochează datele culese din mai multe zile), parametrul este calculat cu ajutorul celuilalt serviciu folosit
* bbox: bounding box-ul calculat mai sus
* height: înălțimea imaginii
* width: lățimea imaginii

Înălțimea și lățimea imaginii sunt calculate cu ajutorul funcției:

Funcția returnează distanța dintre două puncte definite în sistemul epsg4326 exprimată in metrii. Astfel pentru a obține de la serviciul terrascope o imagine cu dimensiuni proporționale cu dimensiunea reală a parcelei definită de bounding box-ul respectiv, parametrii height și width sunt dați de dimensiunile celor 2 laturi exprimate în metri. Un pixel din imagine va reprezenta aproximativ 1 de metru pătrat.

def getDistanceFromLatLonInM(lat1, lon1, lat2, lon2):

R = 6371

dLat = deg2rad(lat2 - lat1)

dLon = deg2rad(lon2 - lon1)

a = math.sin(dLat / 2) \* math.sin(dLat / 2) + math.cos(deg2rad(lat1)) \* math.cos(deg2rad(lat2)) \* math.sin(dLon / 2) \*math.sin(dLon / 2)

c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1 - a))

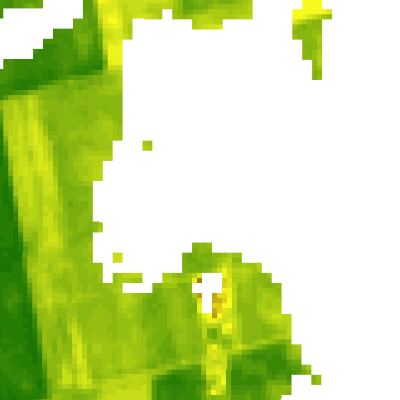
d = R \* c

float(d)

d \*= 1000

return d

În figura urmatoare returnată de serviciul terrascope se poate observa că în unele cazuri imaginea cea mai recentă pentru suprafața dorită este neclară din cauza norilor.



**Serviciul pentru acuratețea imaginilor**

Datele ce ar rezulta din analiza unei astfel de imagini, sunt irelevante deoarece o bucata prea mare din parcelă nu este vizibilă. Pentru a evita această situație sunt folosite datele obținute din serviciul <https://services.terrascope.be/timeseries/v1.0/ts/TERRASCOPE_S2_NDVI_V2/geometry> de tip POST. Request-ul primește 2 parametrii: startDate și endDate și un body de tip json ce conține date referitoare la bucata de pamănt pentru care sunt solicitate informatiile. Serviciul retunează detalii despre imaginile pentru aria de pamănt capturate în perioada definită de startDate și endDate. De exemplu pentru startDate=2022-04-31 și endDate=2022-05-31 și body-ul:

{

"type": "Feature",

"geometry": {

"type": "Polygon",

"coordinates": [[

[27.199243, 45.910026],

[27.209468, 45.911885],

[27.209607, 45.906525],

[27.200563, 45.904793],

[27.199243, 45.910026]

]]

}

}

care corespunde parcelei din imaginea de mai sus, pentru fiecare zi din intervalul definit pentru care există o fotografie cu parcela este returnat numărul total de pixeli, numărul de pixeli vizibili și media indicelui NDVI pentru pixeli. Mai jos se poate observa o parte din răspuns:

"results": [

{

"result": {

"totalCount": 10214,

"validCount": 8715,

"average": 0.5399646586345382

},

"date": "2022-04-30"

},

{

"result": {

"totalCount": 10214,

"validCount": 10214,

"average": 0.6117971411787744

},

"date": "2022-05-05"

},.........

Pe baza acestor date se alege ziua a cărei imagine va fi analizată: se va selecta cea mai recentă dată în care raportul dintre numarul valid de pixeli si numărul total de pixeli este mai mare ca 0.8. Dacă nu se gasește o imagine satisfăcătoare in ultimele 30 de zile de la data curentă se vor adăuga 30 de zile la data de start până se gasește o imagine ce indeplinește criteriul.

Funcția *postPixelCountService(polygonCoordinatesString, days)* apelează serviciul terrascope pentru acuratețea imaginilor. Data selectată este returnată și folosită ca parametru în apelul pentru serviciul de imagini NDVI.

def getOptimalDate(polygonCoordinatesString):  
 ok = 0  
 days = 30  
 max = 0  
 maxDate = 0  
 polygonCoordinatesString = polygonCoordinatesString.replace(" ","")  
 while ok == 0:  
 responseJson = postPixelCountService(polygonCoordinatesString, days)  
 maxDate = responseJson['results'][len(responseJson['results'])-1]['date']  
 if float(responseJson['results'][0]['result']['validCount']) != 0:  
 max = float(responseJson['results'][0]['result']['validCount'])/float(responseJson['results'][0]['result']  
 ['totalCount'])  
 maxDate = responseJson['results'][0]['date']  
 else:  
 max = 0  
 for i in range(len(responseJson['results'])):  
 validRatio = float(responseJson['results'][i]['result']['validCount'])/float(responseJson['results']  
 [i]['result']['totalCount'])  
 if validRatio != 0 and validRatio >= max:  
 max = validRatio  
 maxDate = responseJson['results'][i]['date']  
 if max > 0.8:  
 ok = 1  
 days += 30  
 outputStringList = []  
 outputStringList.append(maxDate)  
 outputStringList.append(max)  
 return outputStringList

**Decuparea imaginii**

Următorul pas este decuparea imaginii. Serviciul terrascope returneaza o imagine dreptunghiulară ce corespunde bounding box-ului în care se încadreaza parcela. Pentru a analiza doar parcela este nevoie să decupăm imaginea transformând culoarea pixelilor ce nu fac parte din parcelă în negru. De exemplu:

**Maparea punctelor pe imagine**

În primul rând este nevoie de a face legătura între coordonatele punctelor ce determină parcela și indicii pixelilor din imagine ale acestor puncte.

def mapPolygonPointsOnImage(bbox, polygonCoordinates, heigth, width):  
 bbox = verifyOrderOfBboxCoordinates(bbox)  
 bbox = epsg3857ToEpsg4326(bbox)  
 bboxPoint1 = [bbox[0], bbox[1]]  
 bboxPoint2 = [bbox[2], bbox[3]]  
 originPoint = [bbox[0], bbox[3]]  
 pixelPositions = []  
 values = pixelMapValue(bbox[0], bbox[2], bbox[1], bbox[3], heigth, width)  
 for i in range(0, len(polygonCoordinates), 2):  
 pixelposx = abs(polygonCoordinates[i] - originPoint[0]) / values[0]  
 pixelposy = abs(polygonCoordinates[i + 1] - originPoint[1]) / values[1]  
 pair = [pixelposx, pixelposy]  
 pixelPositions.append(pair)  
 return pixelPositions

Funcția returnează pozitiile punctelor în matricea de pixeli ce definește imaginea (o imagine este definită de o matrice de pixeli cu dimensiunile egalu cu dimensiunea imaginiil; fiecare pixel are o valoare în spațiul de culori RGB, adică 3 valori din intervalul [0, 255]). Pentru a afla indicii pixelilor punctelor sunt împărțite distanțele de la punctul din stânga jos al pozei (originPoint) cu valorile returnate de următoarea metodă:

def pixelMapValue(xmax, xorigin, ymax, yorigin, heigth, width):  
 xdif = abs(xmax - xorigin)  
 # print(xdif)  
 ydif = abs(ymax - yorigin)  
 # print(ydif)  
 pixelValueX = xdif / width  
 pixelValueY = ydif / heigth  
 values = [pixelValueX, pixelValueY]  
 return values

Este calculată distanța dintre originea (stânga jos) și punctul din dreapta sus a imaginii pe cele 2 direcții. Apoi valorile sunt impărtite la dimensiuni.

Decuparea are loc in metoda **cropImage**:

def cropImage(imagePath, pixelIndicesArray):  
 img = cv2.imread(imagePath)  
 pixelIndicesArray = roundFloatList(pixelIndicesArray)  
 pts = np.array(pixelIndicesArray)  
  
 # (1) Crop the polygon  
 polygon = cv2.boundingRect(pts)  
 x, y, w, h = polygon  
 croped = img[y: y + h, x:x + w].copy()  
  
 # (2) make mask  
 pts = pts - pts.min(axis=0)  
  
 mask = np.zeros(croped.shape[:2], np.uint8)  
 cv2.drawContours(mask, [pts], -1, (255, 255, 255), -1, cv2.LINE\_AA)  
  
 # (3) do bit op  
 dst = cv2.bitwise\_and(croped, croped, mask=mask)  
  
 # (4) add white background  
 bg = np.ones\_like(croped, np.uint8) \* 255  
 cv2.bitwise\_not(bg, bg, mask=mask)  
 dst2 = bg + dst  
  
 cv2.imwrite("resources/images/croped.png", croped)  
 cv2.imwrite("resources/images/mask.png", mask)  
 cv2.imwrite("resources/images/dst.png", dst)  
 cv2.imwrite("resources/images/dst2.png", dst2)

Mai întai este decupată imaginea pe înălțime și lătime cu ajutorul metodei boundingRect() din librăria OpenCv. Metoda primește ca argument un array de puncte si apoi returneaza originea de unde trebuie decupată imaginea, înălțimea și lătimea pană la care trebuie decupată. În general imaginile nu ar trebui sa fie decupate pe înălțime sau lungime pentru că ele sunt deja create pe baza unui boundingbox.

În continuare, este creată o mască. Creăm o matrice de pixeli negri de dimensiunea imaginii, apoi cu ajutorul metodei drawContours() a librăriei OpenCv, desenăm o forma albă ce corespunde parcelei. Metoda primește ca parametrii imaginea sursă (în cazul de față masca), punctele formei (punctele identificate anterior ), parametrul care indică ce contururi trebuie desenate (-1 înseamnă ca vor fi desenate toate), culoarea conturului (255,255,255 pentru alb), parametrul ce indică daca trebuie colorat și interiorul conrului (-1 și orice altă valoare negativă indică faptul ca va fi colorat și interiorul) și tipul de linie (cv2.LINE\_AA este o linie care face trecerea între fundal si formă sa fie graduală). În figura MASXA se poate observa masca pentru FIGURILE DE MAI SUS.

Folosind metoda bitwise\_and aplicăm masca peste imagine. Metoda poate fi folosită și pentru a suprapune 2 imagin, dar pentru că este nevoie doar de funcționalitatea de aplicare a măștii primii 2 parametrii sunt imaginea ințială iar cel de-al treilea este masca. Mai departe este adăugat un fundal alb. În aplicație este folosită poza decupată cu fundal negru, cea cu fundal alb a fost creată pentru a vedea dacă afectează analiza. Rezultatele sunt identice.

**Separarea culorilor**

Pasul următor este separarea culorilor. Scopul este de a împărți imaginea pe 3 culori. Cele 3 culori au fost alese în funcție de indicele NDVI: zona verde(vegetație satisfăcătoare), zona galbenă(vegetație uscată) și zona maro (vegetatie foarte uscată, drumuri sau clădiri).

def colorMask(imagePath, color):  
 ## Read  
 try:  
 img = cv2.imread(imagePath)  
 except:  
 logging.exception("Could not read image from relative path " + imagePath)  
  
 ## convert to hsv  
 hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)  
  
 if color.upper() == "GREEN":  
 mask = cv2.inRange(hsv, (41, 25, 25), (70, 255, 255))  
 elif color.upper() == "YELLOW":  
 mask = cv2.inRange(hsv, (23, 25, 25), (40, 255, 255))  
 elif color.upper() == "BROWN":  
 mask = cv2.inRange(hsv, (13, 25, 25), (22, 255, 255))  
  
 ## slice the color  
 imask = mask > 0  
 color = np.zeros\_like(img, np.uint8)  
 color[imask] = img[imask]  
  
 ## save  
 cv2.imwrite(f"resources/images/{color}.png", green)

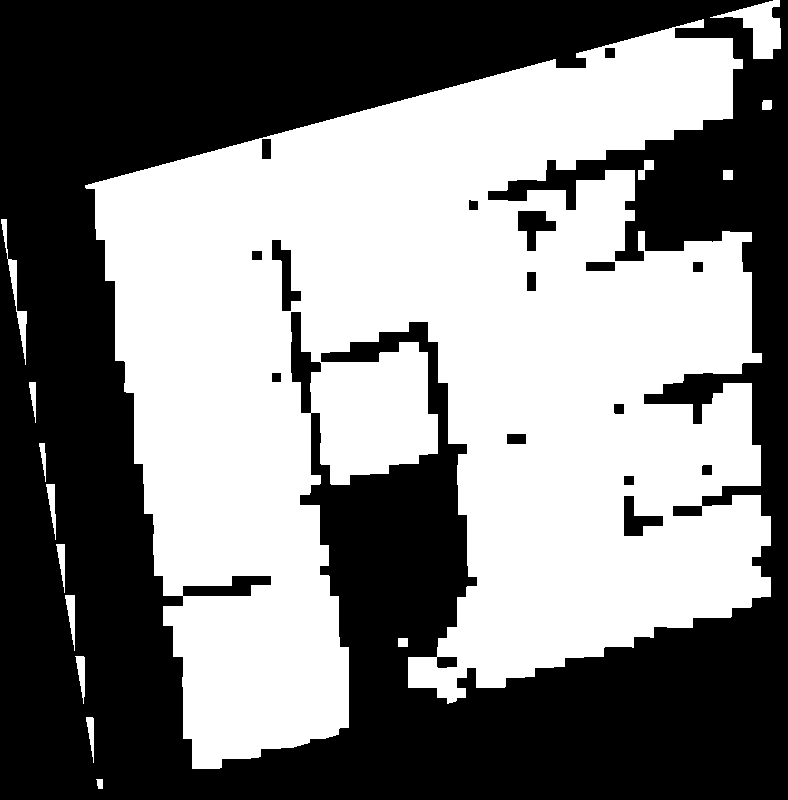
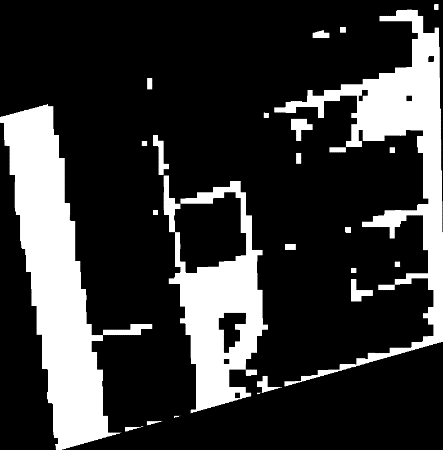
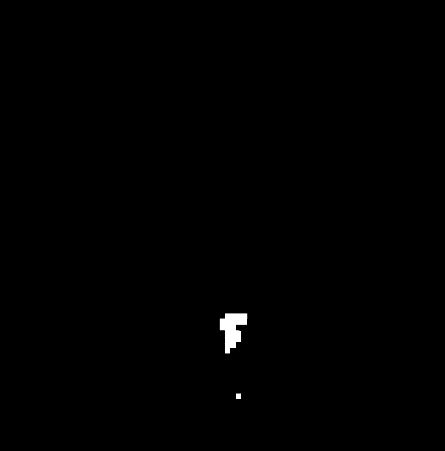
Primul pas este trecerea imaginii în spațiul de culoare HSV. Imaginea care este citită este în spațiul de culoare RGB (red, green, black) dar prin folosirea metodei cv2.imread() imaginea este trecută în spațiul de culoare BGR (black, green, red). Pentru a realiza transformarea, este folosită funcția cv2.cvtColor ce primește ca parametrii imaginea citită cu ajutorul metodei din librărie și tipul de transformare, în cazul de fată din spațiul BGR în spațiul HSV.

**Spațiul HSV**

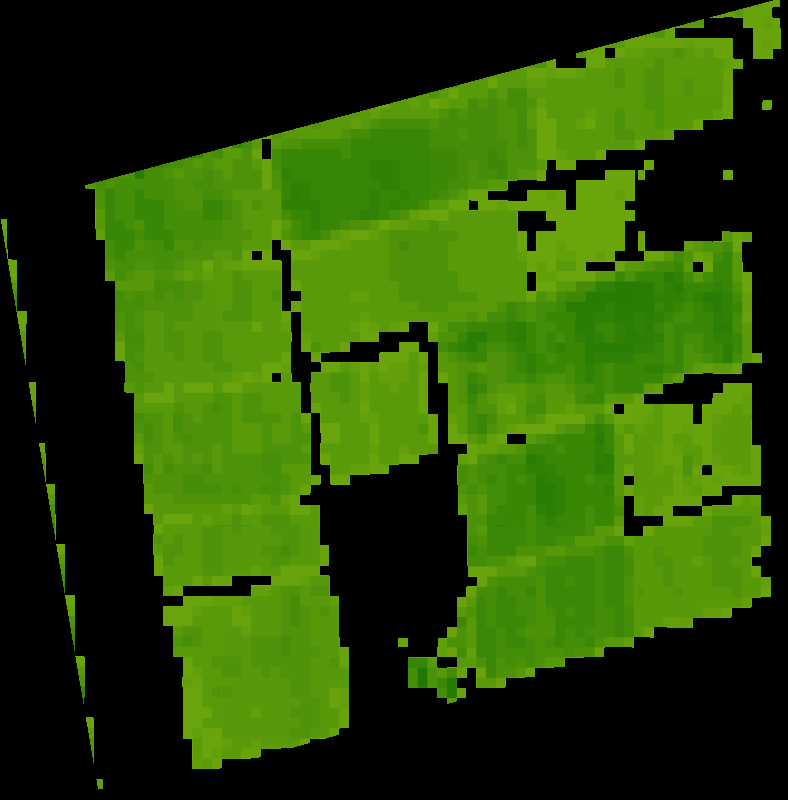
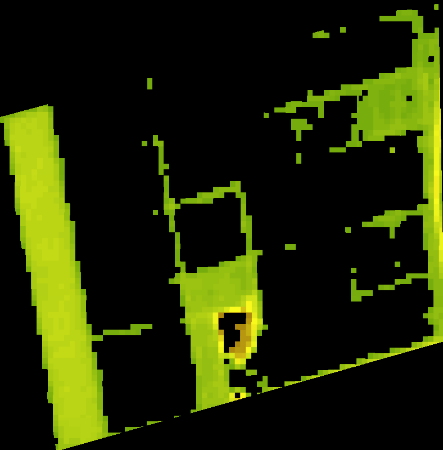
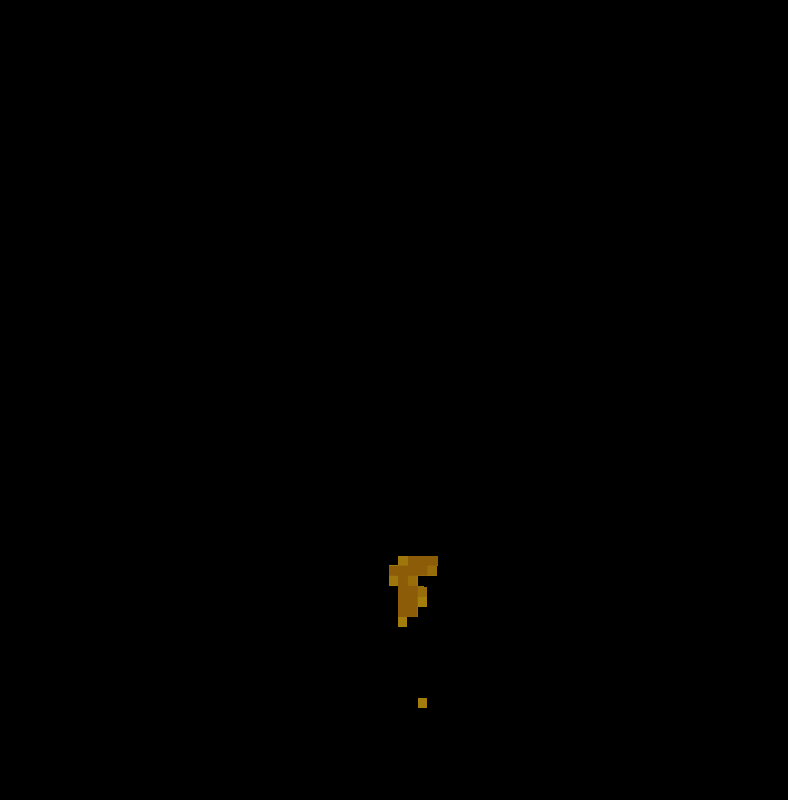
HSV este un spațiu de culoare ce are 3 componente:

* Hue, specifică unghiul culorii pe cercul de culori RGB. Poate lua valori între 0 și 179 în OpenCV.
* Saturation, controlează cantitatea de culoare. O valoare ce se apropie de 100% va aduce cea mai pură varianta a culorii, în timp ce o valoare apropiată de 0% va aduce o culoare gri. Poate lua valori între 0 și 255 în OpenCV.
* Value controlează luminozitatea culorii. O culoare cu value aproape de 0% este de fapt negru pur în timp ce o culoare cu value aproape de 100% nu va conține negru. Poate lua valori între 0 și 255 în OpenCV.

În urma rulării algoritmului pentru mai multe valori am ajuns la următoarele intervale HSV pentru cele 3 culori: Verde (41, 25, 25)-(70, 255, 255); Galben (23, 25, 25)-(40, 255, 255); Maro: (13, 25, 25)-(22, 255, 255). Funcția cv2.inRange primește 3 parametrii: imaginea și cele 2 limite între care trebuie sa se situeze culoarea în spațiul HSV. Output-ul este este o mască binară în care pixelii care se încadrează în interval sunt albi iar pixelii care nu se încadrează sunt negri.

După crearea măștilor, este creată matricea de pixeli imax ce are ca valori True si False. True pentru pixelii albi și False pentru pixelii negri. Apoi este creată o imagine nouă, color, de dimensiunile imaginii inițiale care inițializata cu toți pixelii negri. Apoi folosind matricea imask, pixelii din matricea imask care au valoarea True (sunt albi deci se află în intervalul culorii) copiază valoarea din imaginea inițială (ce corespunde imaginii din input) în imaginile cu cele 3 culori.

**Determinarea poligoanelor**

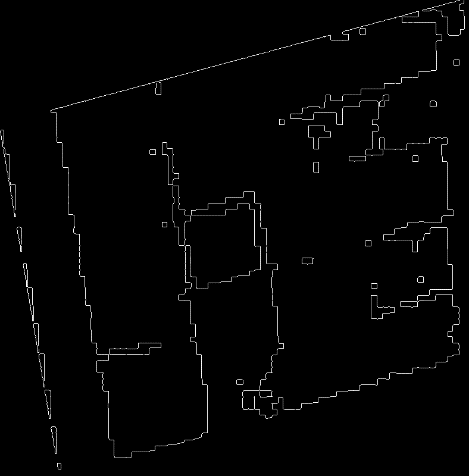
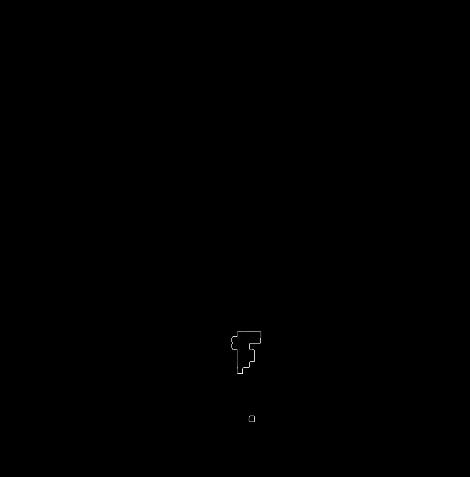
Procesul de determinare al poligoanelor are loc pentru fiecare dintre cele 3 cateogrii NDVI sau pentru cele 3 culori verde galben și maro.

Pentru a produce un output cu date care să ușureze analizarea terenului nu este de ajuns doar impărtirea imaginii pe categorii de culoare. Aceste zone trebuie localizate prin coordonate reale. În exemplul de mai sus se poate oberva că zonele pot fi reprezentate de mai multe suprafețe separate de pămănt, deci este nevoie de identificarea acestor poligoane și a coordonatelor.

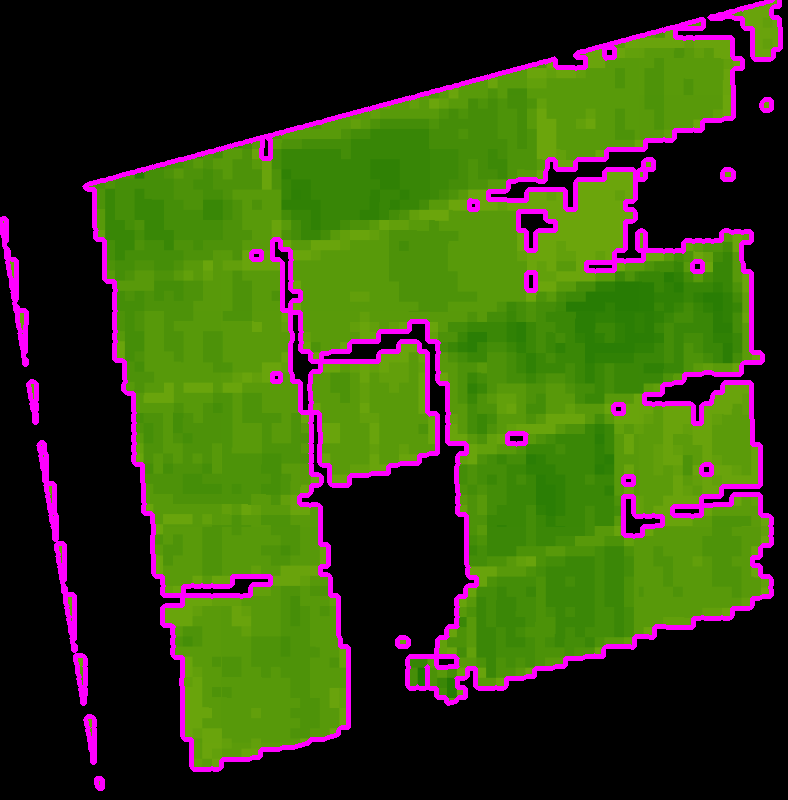
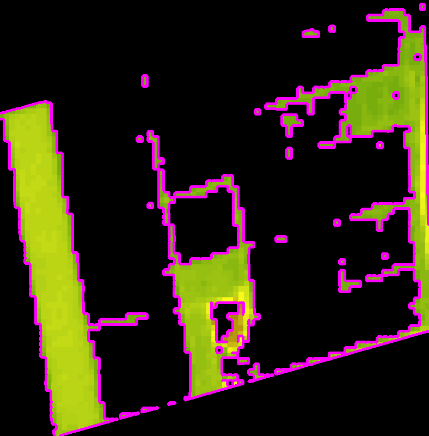
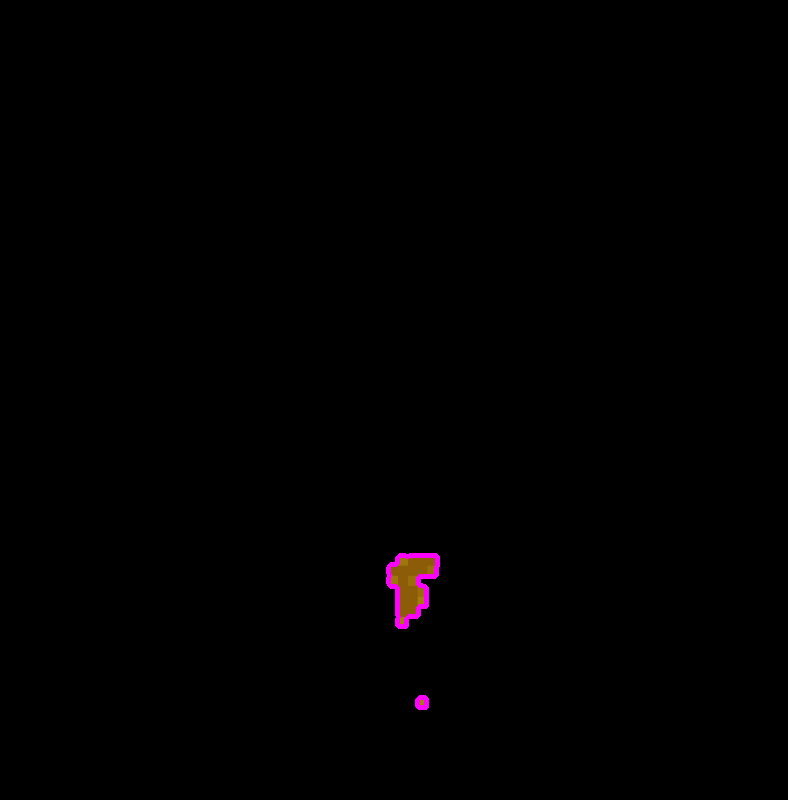
**Identificarea contururilor**

def findContours(image):  
 gray = cv2.cvtColor(image,cv2.COLOR\_BGR2GRAY)  
 edged = cv2.Canny(gray,100,600)  
 contours, \_ = cv2.findContours(edged,cv2.RETR\_EXTERNAL,cv2.CHAIN\_APPROX\_NONE)  
 return contours

Primul pas este identificarea contururilor. Primul pas este transformarea imaginii în grayscale folosind funcția cv2.cvtColor(), cu parametrul COLOR\_BGR2GRAY ce specifică transformarea din spațiul de culoare standard din OpenCv BGR în Grayscale. Apoi este aplicată metoda de detectare a marginilor Canny. Marginea este o discontinuitate în intensitatea pixelilor. Funcția din librăria OpenCv primește ca input imaginea grayscale și 2 argumente: limita de jos (pixelii cu intensitatea mai mică nu pot fi margini) și limita de sus (pixelii cu intensitatea mai mare sunt considerati automat margini).

În imaginile din output în alb-negru, pixelii pot avea doar 2 valori alb sau negru deci imaginile se pot fi caracterizate ca fiind imagini binare (imaginile binare sunt imagini în care pixelii pot avea doar 2 valori). Metoda cv.finContours() funcționează cel mai bine pentru imagini binare. Metoda de detectare este RERT\_EXTERNAL care returneaza doar contururile exterioare deoarece contururile interioare reprezintă poligoane de altă culoare sau poligoane care se află în acceași categorie dar diferă intensitatea culorii. Parametrul CHAIN\_APPROX\_NONE, reprezintă metoda de aproximare folosită. În cazul de față, pentru un output mai detaliat nu este folosită nicio metodă de aproximare. Totuși dacă output-ul ar fi testat pe teren de utilizatori există posibilitatea de a schimba parametrul în CHAIN\_APPROX\_SIMPLE pentru un output mai puțin detaliat dar mai ușor de analizat. Output-ul este o listă de contururi, un contur la randul său conține o listă de puncte. Desenate peste imagini arată in felul următor:

 ** **

**Separarea în poligoane**

După cum se poate observa în imagini contururile sunt reprezentate de foarte multe puncte. Output-ul final trebuie sa contină doar punctele cele mai importante nu întreg conturul. Pentru a determina colturile imaginilor este folosită metoda cv2.CornerHarris. Colțurile unei imagini sunt regiunile în care sunt variații mari în gradientul imaginii în toate dimensiunile si direcțiile posibile. Gradientul este o schimabare direcțională în intensitatea sau culoarea unei imagini.

def extractPolygonCorners(imagePath, color):  
 img = cv2.imread(imagePath)  
 hsv = cv2.cvtColor(img, cv2.COLOR\_BGR2HSV)  
 hsv = np.float32(hsv)  
 if color.upper() == "GREEN":  
 hsv = cv2.inRange(hsv, (41, 25, 25), (70, 255, 255))  
 elif color.upper() == "YELLOW":  
 hsv = cv2.inRange(hsv, (23, 25, 25), (40, 255, 255))  
 elif color.upper() == "BROWN":  
 hsv = cv2.inRange(hsv, (13, 25, 25), (22, 255, 255))  
  
 dst = cv2.cornerHarris(hsv, 5, 3, 0.04)  
   
 # thres poate fi modificat pt a gasi mai multe colturi  
 thres = 0.1 \* dst.max()  
   
 img[dst > thres] = [0, 0, 255]  
  
 cv2.imwrite('resources/images/cornerHarris.png', img)  
  
 corners = []  
 for i in range(len(img)):  
 for j in range(len(img[0])):  
 if (img[i][j] == [0, 0, 255]).all():  
 corners.append([j, i])  
  
 #cv2.imshow('corners', img)  
 #cv2.imwrite(f'Imagini/{color}points.png', img)  
 return corners

Metoda primește la intrare imaginea care va fi analizată,

Funcția nu returnează puncetle găsite ci o matrice 2d de aceeași dimensiune ca imaginea din input cu probabilități. Fiecare poziție din matrice conține o valoare numerică ce reprezintă probabilitatea ca pixelul de pe acea pozitie sa fie un colț. Este nevoie de un threshold după care vor fi selectate punctele. În cazul de față valoarea este 0.1 din valoarea maximă a matricei din output-ul metodei Corner Harris. Fiecare punct ce trece de treshold este considerat colț și este transformat în culoare roșie în imaginea inițială din spațiul de culoare BGR, adică primește valoarea [0, 0, 255]. Apoi, pe baza acestei imagini este creată o listă de perechi care conține indicii colțurilor. Folosind doar Corner Harris este posibil ca serviciul să returneze colțurile trecănd peste detectarea contururilor. Dar pentru a repartiza colțurile pe poligoane este nevoie de ambele rezultate.

În metoda extractPolygons ce primește ca input contururile și colțurile imaginii sunt create poligoanele pentru output. Contururile conțin toate punctele ce separa poligoanele (inclusiv colțurile) grupate. Acum datorită faptului că colțurile sunt cunoscute pot fi create poligoane. Un poligon este construit pe baza unui contur din care au fost extrase doar punctele care sunt colțuri.

def extractPolygons(contours, corners):  
 polygons = []  
 for contour in contours:  
 poly = []  
 for coords in corners:  
 if coords in contour:  
 poly.append(coords)  
 polygons.append(poly)  
  
 return polygons

Pe baza acestor poligoane este creată o listă de obiecte de tip Poligon.

class Polygon:  
   
 def \_\_init\_\_(self, coords, code):  
 self.coords = coords  
 self.code = code

Poligonul conține o listă de puncte și un cod ce corespunde unei culori.

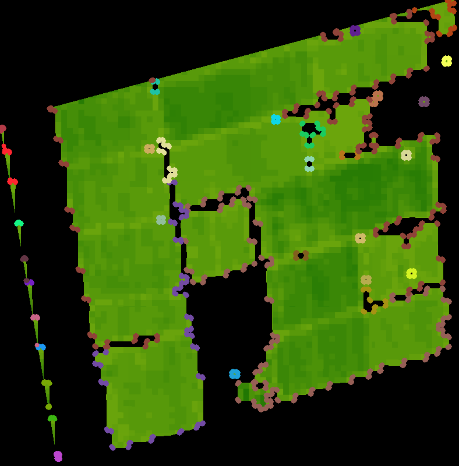
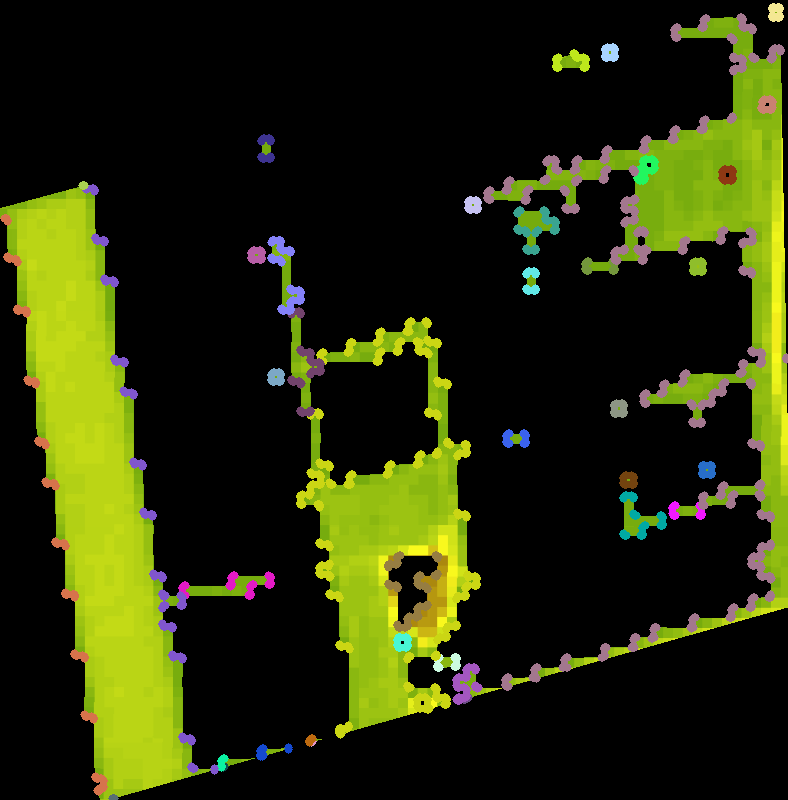
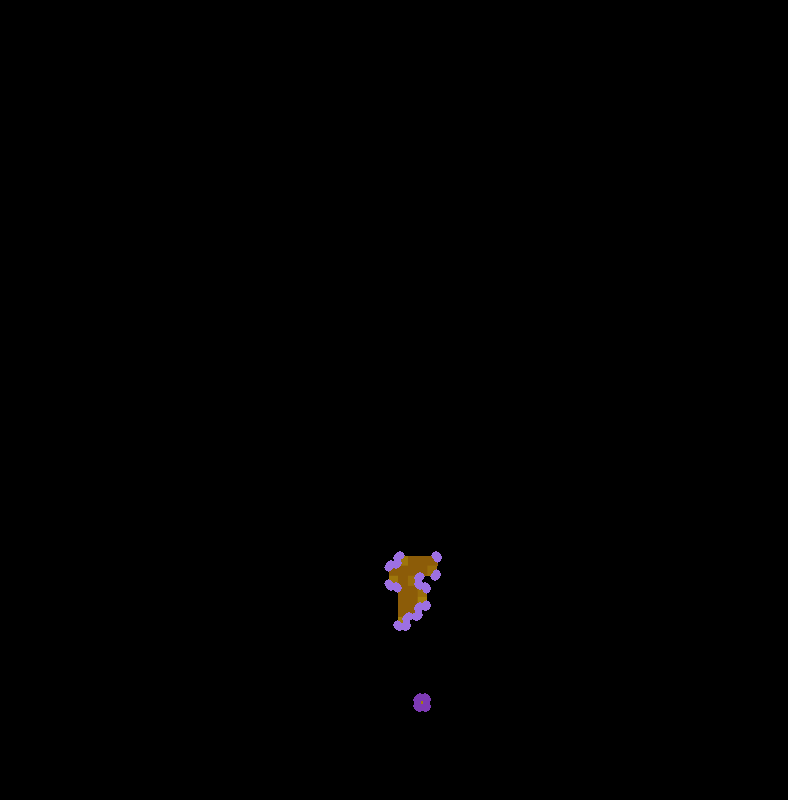
for poly in polygonCoords:  
  
 # Candidates that have less than 3 points cannot be polygons.  
 if len(poly) < 3:  
 continue  
  
 coords = pixelsIndicesToCoordinates(poly, height, width, coordinatesBBOX)  
 p = Polygon(coords, colorCode[color.lower()])  
 polygonList.append(p)  
  
logging.info("Polygons extracted from the " + color + " image.")

Un poligon trebuie sa conțină cel puțin 3 puncte pentru a fi considerat la output. Apoi folosind metoda pixelsIndicesToCoordinates punctele exprimate în indici sunt transformate în puncte de coordonate, este creat poligonul și adăgat la lista finală.

Din cauza modului în care sunt determinate contururile, în unele cazuri poligoanele nu sunt determinate optim. Un contur se termină la marginea imaginii sau cănd se intersectează în trasarea sa cu un punct ce îi apartine. Sunt cazuri în care punctele poligoanelor nu determină o zonele corect din cauza unor margini ale imaginii sau din cauza ateției foarte mari la detalii a metodei care găseste contururile și din cauza marginilor. O soluție are fi extinderea celor 4 laturi ale imaginilor cu pixeli negri pentru a elimna întreruperile nedorite provocate de margini. În urma analizări mai multor seturi de date am obervat că rezultatul ar putea fi imbunătățit prin eliminarea unor pixeli ce unesc mase mai mari de culoare. La întalnirea unor astfel de puncte desenarea conturului se întoace, întâlnește un punct ce îi aparține si se termină, ceea ce duce la crearea unor poligoane care nu sunt închise. Pierderea de acuratețe ar fi foarte mică și ar putea îmbunătăți calitatea output-ului. Pixelii ar fi ‘eliminați’ prin colorarea lor în culoarea negru.

În imaginile urmatoare, ce au foarte multe variații de culoare, pot fi observate și cazuri extreme ce duc la crearea unor poligoane ce ramân deschise. Scopul este ca majoritatea poligoanelor să fie închise. Chiar și în cazuri extreme, output-ul este relevant pentru parcelă intrucât punctele tot reprezintă locuri în care este o variație a vegetației. Fiecare poligon are punctele colorate cu o culoare aleatoare pentru a putea face distincția între poligoane.

 **** 

**Output**

Pasul final este generarea unui json folosind poligoanele obținute și statisticile. Json-ul conține un atribut ‘map’ cu datele referitoare la puncte și un atribut ‘statistics’.

"map": {  
 "crs": {  
 "name": "urn:ogc:def:crs:EPSG::4326",  
 "type": "name"  
 },  
 "features": [  
 {

"geometry": {  
 "coordinates": [  
 [  
 45.91813587640448,  
 27.200360944162437  
 ],  
 [  
 45.91813587640448,  
 27.200453010152284  
 ],  
 [  
 45.91814473033706,  
 27.200347791878173  
 ],  
 [  
 45.918153584269646,  
 27.200479314720813  
 ]  
 ],  
 "type": "Polygon"  
 },  
 "id": 1,  
 "properties": {  
 "classnr": 2  
 }

…..

}

Atributul ‘crs’ conține date referitoare la sistemul de coordoante în care sunt reprezentate punctele

**Generarea statisticilor**

În plus față de punctele grupate în poligoane serviciul, json-ul returnat de serviciul de clasificare conține și o secțiune de statistici. În această sectiune este notat, pentru toate cele 3 culori și pentru pixelii acoperiți de nori, raportul dintre numarul de pixeli de culoare și numărul total de pixeli. Fiecare categorie are asociat un cod și un hex de culoare :

* Codul -1 și hex-ul #FFFFFF pentru pixelii albi/acoperiți de nori
* Codul 0 și hex-ul #8A7010 pentru pixelii maro
* Codul 1 și hex-ul #8A6210 pentru pixelii galbeni
* Codul 2 și hex-ul #768A10 pentru pixelii verzi

"statistics": [  
 {  
 "code": 0,  
 "colorHex": "#8A7010",  
 "coverage": 0.007119808197484246  
 },  
 {  
 "code": 1,  
 "colorHex": "#8A6210",  
 "coverage": 0.2173989761415845  
 },  
 {  
 "code": 2,  
 "colorHex": "#768A10",  
 "coverage": 0.7754812156609313  
 },  
 {  
 "code": -1,  
 "colorHex": "#FFFFFF"

"coverage": 2.2056309758814254e-06  
 }]

**Baza de date**

**Introducere**

Din cauza timpul de răspuns foarte mare al componentei de clasificare este nevoie de stocarea output-urilor. Componenta va fi apelată pentru analiza inițială a unei parcele sau dacă se dorește efectuarea unei noi analize. Altfel, rezultatele vor fi obținute din baza de date. Baza de date este distribuită de tip NoSql și este construită cu ajutorul framework-ului Hadoop.

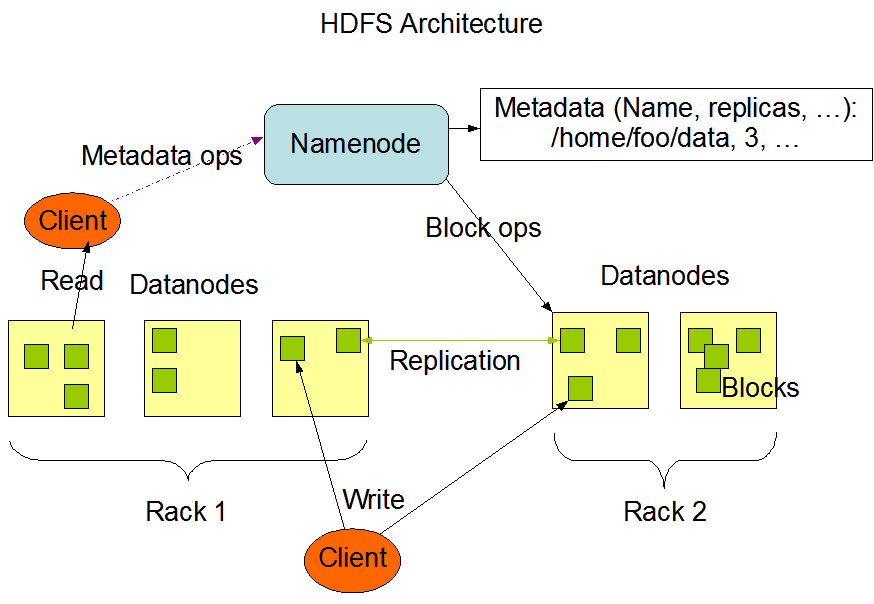
**Hadoop**

Hadoop (High Availability Distributed Object Oriented Programming) este o plaformă software open-source construită cu Java ce se ocupă cu managementul, stocarea și procesarea volumelor mari de date. Hadoop are la bază un cluster de computere. Un cluster este un set de computere ce lucrează impreună ca un singur sistem. Performanța computerelor poate varia de la mașini extrem de performante până la computere normale sau chiar puțin performante. Performanța computerelor influențează direct puterea sistemului. Cluster-ul poate fi alcătuit din orice număr de computere și poate fi extins în orice moment. Componentele de bază Hadoop sunt Hdfs (Hadoop distributed file system) și modelul de programare Map Reduce.

**Hdfs**

Hdfs este un sistem de date distribuit conceput sa ruleze pe hardware standard. Hdfs, spre deosebire de majoritatea sistemelor de date, este foarte tolerant la erori și poate fi folosit pe hardware low-cost. Un sistem de date HDFS poate fi format din sute sau mii de computere fiecare stocând o parte din date. Hdfs imparte datele în blocuri și le stochează pe computerele clusterului Fiecare componenentă are posibilitatea de a eșua așa că Hdfs are un mod foarte eficient de a detecta problemele de hardware si de a le rezolva automat.

Hdfs are o arhitectură de tip master/slave. Nodul master este unic și se numește NameNode, nodurile slave se numes DataNodes. În general fiecare computer din cluster conține un nod de date dar este posibil să conțină și mai multe. Pot fi adăugate noduri de date în orice moment iar Hdfs va realoca datele pentru cât mai optim. Name node este cel care lansează toate operațiunile de lucru cu datele: citire, scriere, duplicare, deschidere, închidere etc. Orice proces solicitat de client se face prin intermediul namenode-ului. Mai departe, name node-ul împarte task-ul în sub task-uri care apoi sunt trimise către data node-uri pentru a fi rulate. Singura excepție este stocarea datelor unde clientul poate lăsa name node să distribuie datele sau le poate scrie direct în data node-uri. Din punctul de vedere al procesării, totul are loc la nivelul data node-urilor. Din puntul de vedere al stocării datelor, datele propriu-zise sunt împărțite în blocuri și stocate în data node-uri, în timp ce name node-ul conține meta-date.

****

Sursă imagine https://hadoop.apache.org/

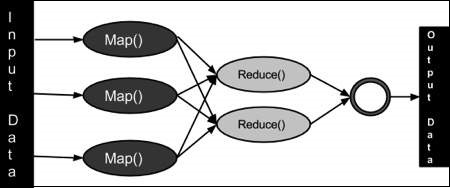
**MapReduce**

Map reduce este o tehnică de procesare pentru computarea distribuită. Algoritmul conține 2 părți map și reduce. Map primește un set de date si îl transformă în alt set de date în care fiecare element este o pereche cheie-valoare. Reduce primește la intrare output-ul funcției de map și combină perechile într-un set mai mic de perechi cheie-valoare. Cel mai mare avanaj este că procesarea datelor poate fi scalată pe mai multe noduri. Odată ce o aplicație map reduce este scrisă rularea sa pe sute, mii sau zeci de mii de computere este doar o schimbare de configurare.

MapReduce se execută in 3 pași: map, shuffle și reduce

* Map – Scopul este de a procesa datele de intrare. În general datele de intrare sunt de forma unui fișier sau director stocate in HDFS. Fisierul de intrare este transmis către mapper linie cu linie. Mapper-ul procesează datele și crează bucați mai mici de date.
* Reduce – combină pașii de shuffle și reduce. Procesează datele rezultate din mapper. Operațiile și logica aplicatiei de map reduce este scrisă la cest nivel. Noul output este stocat în HDFS.

Procesul de map reduce este recepționat la name node care trimite procesul mai departe către data nodes care vor rula procesul în paralel pe datele lor. Output-ul final este reuniunea rezultatelor de pe nodurile de date.



Sursă https://www.tutorialspoint.com/hadoop

**Hbase**

Hbase este sistem de management al bazelor de date non relațional orientat pe coloane ce rulează peste HDFS. Hbase este o metodă toelrantă la erori de avea acces de scriere și citire aleatoare la volume mari de date. Spre deosebire de sistemele realționale de baze de date, Hbase nu are un limbaj de interogare precum SQL deoarece Hbase este scris în Java. Pentru a adăuga posibilitatea de a face interogări asupra este nevoie de a folosi Hive împreună cu Hbase.

Un sistem Habse scalează liniar. Conține un set de tabele cu rânduri și coloane la fel ca o bază de date tradițională. Fiecare tabelă trebuie să aibă un o cheie primară. Toate operațiile de acces la tabele Hbase trebuie să folosească această cheie primară.

**Ambari**

**Mașina virtuală**

**4. Bibliografie**

<https://gisgeography.com/ndvi-normalized-difference-vegetation-index/>

<https://custom-scripts.sentinel-hub.com/sentinel-2/ndvi/>

<https://pillow.readthedocs.io/en/stable/>

<https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_setup/py_intro/py_intro.html#intro>

<https://docs.terrascope.be/#/README>