

## ▼ Implementación y agregación de segmentadores

### 1. Integrantes del grupo

Este trabajo pertenece al Grupo 18 - Equipo 3. Los integrantes somos:

- Irene Fanjul i Penella
- Miguel Jiménez Gomis
- Carles Serra Vendrell
- Adrián Bañuls Arias
- Miriam Pardo Cuesta

Enlace a Google Colaboratory: <https://colab.research.google.com/drive/1I7sQRmyx9-ks8E7igUie067IKDGkNXow?usp=sharing#scrollTo=GCFdKnFBXbUV>

### 2. Organización de las reuniones

Reuniones	Descripción
14/02/2021	Planificación de las tareas a realizar. Elección del segmentador a desarrollar.
18/02/2021	Puesta en común de las soluciones investigadas de forma independiente por cada integrante y de las dudas surgidas.
23/02/2021	Corrección de bugs y análisis de los resultados.
24/02/2021	Redacción final de la entrega y limpieza del notebook.

Todos los integrantes del grupo han participado en todas las reuniones.

### 3. Descripción del problema

Como problema a tratar en este trabajo se ha seleccionado la detección de barcos en imágenes satélite. Se han escogido un subset de imágenes del Dataset de la competición de Kaggle de Airbus [2], la cual incluye el Ground truth de las imágenes para poder evaluar los resultados obtenidos.

### 4. Solución propuesta

Se han implementado 3 segmentadores de imágenes que generan máscaras binarias indicando las posiciones en las que puede haber un barco. Así mismo, se ha implementado un cuarto segmentador que combina las salidas de los 3 anteriormente mencionados agregando sus resultados y codificando un método de selección en caso de empate.

Se importan las librerías necesarias y se cargan los datos de prueba. Por falta de espacio no se muestran en este documento, se pueden ver en el Notebook entregado o en el enlace de Google Colab proporcionado

## ▼ Importación de librerías

[ ] ↳ 2 celdas ocultas

## ▼ Programación de los segmentadores

Antes de programar los Segmentadores se definen las variables comunes y el método de evaluación de estos segmentadores para ello primero se genera la función que decodifica el ground truth del dataset.

```
1 # Decodificación del ground truth
2 def decodeGroundTruth(encoded_pixel, img_size, archive_name):
3     encoded_pixel[0].split(' ')
4     rle = list(map(int, encoded_pixel[0].split(' ')))
5     pixel, pixel_count = [], []
6     [pixel.append(rle[i]) if i%2==0 else pixel_count.append(rle[i]) for i in range(0, len(rle))]
7     rle_pixels = [list(range(pixel[i], pixel[i]+pixel_count[i])) for i in range(0, len(pixel))]
8     rle_mask_pixels = sum(rle_pixels, [])
9     mask_img = np.zeros((img_size[0]*img_size[1],1), dtype=int)
10    mask_img[rle_mask_pixels] = 255
11    l,b=cv2.imread(archive_name).shape[0], cv2.imread(archive_name).shape[1]
12    mask = np.reshape(mask_img, (b, 1)).T
```

```

13     #plt.imshow(mask, cmap='gray')
14     return mask/255

```

Una vez se tiene esta función para generar la máscara del ground truth, se procede a desarrollar una función para evaluar los segmentadores. Esta función tiene 2 partes:

- Unbucle que itera por todas las imágenes del conjunto de test y calcula su máscara del segmentador y decodifica su ground truth.
- Una función que dada la máscara y el ground truth calcula la matriz de confusión para cada imagen.

Por ultimo los resultados se agregan y se muestra el promedio de la matriz de confusión.

```

1 def calculate_conf_matrix(mask, ground_Truth):
2     '''Esta función calcula los datos de la matriz de confusión para una máscara
3     generada por un segmentador y su ground truth.
4     '''
5     height, width = mask.shape
6     tp, fp, tn, fn = 0, 0, 0, 0
7     for i in range(height):
8         for j in range(width):
9             if (int(mask[i,j] > 0) and (ground_Truth[i,j] > 0)):
10                 tp += 1
11             elif (int(mask[i,j] > 0) and (ground_Truth[i,j] == 0)):
12                 fp += 1
13             elif (int(mask[i,j] == 0) and (ground_Truth[i,j] == 0)):
14                 tn += 1
15             elif (int(mask[i,j] == 0) and (ground_Truth[i,j] > 0)):
16                 fn += 1
17     return tp, fp, tn, fn
18
19 def test_segmentator(segmentator):
20     ''' Esta función itera por todas las imágenes y calcula el promedio de sus matrices de confusión
21     y muestra las métricas relevantes para un segmentador pasado como parámetro
22     '''
23     ground_Truth_Data = pd.read_csv("groundtruth.csv") #Se cargan los datos
24     tp, fp, tn, fn, esp, acc, rec = [], [], [], [], [], [], []
25     n = 0
26     for row in ground_Truth_Data.iterrows():
27         imageName = row[1][0]
28         img_ = io.imread(imageName)
29         GT = decodeGroundTruth([row[1][1]], img_.shape, imageName) # Se decodifica el Ground truth
30         mask = segmentator(img_) #Se obtiene la máscara
31         result = calculate_conf_matrix(mask, GT)
32         debug = False
33         if debug:
34             fig, (ax1,ax2,ax3) = plt.subplots(1,3);      fig.set_size_inches(10,10)
35             ax1.imshow(img_);      ax1.set_title("Imagen Original")
36             ax1.axis('off');      ax2.imshow(GT, cmap='gray')
37             ax2.set_title("Ground Truth");      ax2.axis('off')
38             ax3.imshow(mask, cmap='gray');      ax3.set_title('Segmentador')
39             ax3.axis('off');      plt.show();
40             tp.append(result[0]);      fp.append(result[1])
41             tn.append(result[2]);      fn.append(result[3])
42             esp.append(tn[n]/(tn[n]+fp[n])) if tn[n] + fp[n] != 0 else esp.append(0.00)
43             acc.append(tp[n]/(tp[n]+fp[n])) if tp[n] + fp[n] != 0 else acc.append(0.00)
44             rec.append(tp[n]/(tp[n]+fn[n])) if tp[n] + fn[n] != 0 else rec.append(0.00)
45             if debug:
46                 print(f"{imageName} Recall: {rec[n]:.2f} Especificidad: {esp[n]:.2f} Precisión: {acc[n]:.2f}")
47             n += 1
48
49     print("Promedio de las métricas:")
50     print("TP: ", np.sum(tp)/n, " FP: ", np.sum(fp)/n, "\nTN: ", np.sum(tn)/n, " FN: ", np.sum(fn)/n)
51     print("Especificidad: ", np.sum(esp)/n, " Precision: ", np.sum(acc)/n, "Recall: ", np.sum(rec)/n)

```

## ▼ Segmentador Kmeans

El algoritmo consiste en asignar cada uno de los n ejemplos uno de los k clusters, donde k es un número definido previamente. El objetivo es minimizar las diferencias entre los grupos de cada cluster y maximizar las diferencias entre clusters. El algoritmo utiliza un proceso heurístico para calcular la solución óptima.

```

1 def apply_kmeans(image, K):

```

```

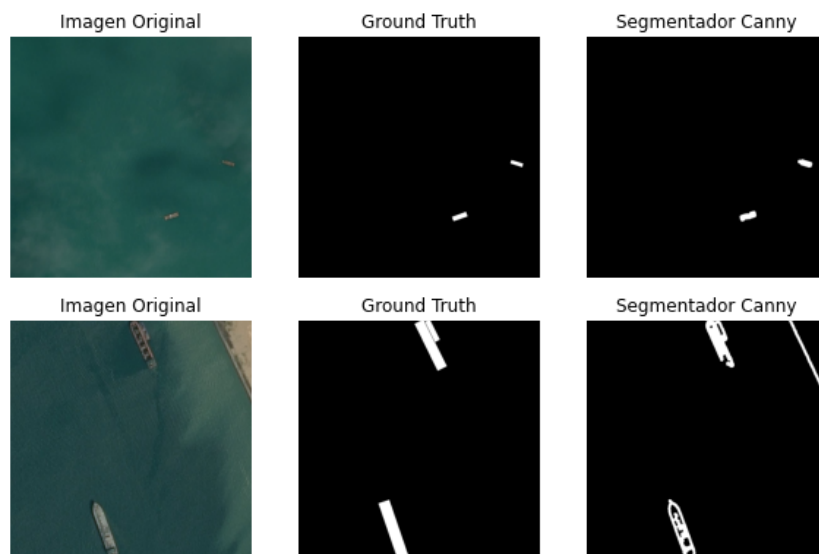
2     image=cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
3     vectorized = image.reshape((-1,3))
4     vectorized = np.float32(vectorized)
5     criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
6     attempts=10
7     ret,label,center = cv2.kmeans(vectorized,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
8     center = np.uint8(center)
9     res = center[label.flatten()]
10    result_image = res.reshape((image.shape))
11    return result_image
12
13 def kmeans_segmentator(img):
14     kmeans = apply_kmeans(img, 10)
15     kmeans_gray = cv2.cvtColor(kmeans, cv2.COLOR_BGR2GRAY)
16     imagen_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
17     kmeans_mask = np.where(kmeans_gray > 70, 0, 255)
18     return kmeans_mask/255
19
20 image1='ejemplo_2.jpeg'
21 img1 = io.imread(image1)
22 image2='ejemplo_1.jpeg'
23 img2 = io.imread(image2)
24
25 kmeans_mask = kmeans_segmentator(img1)
26 en_pix2 = ['518036 2 518800 6 519565 10 520331 12 521099 12 521868 12 522636 12 523404 12 524172 13 524941 12 525709 12 526477 12 527245 12 528013 12 528781 12 529549 12 530317 12 531085 12 531853 12 532621 12 533389 12 534157 12 534925 12 535693 12 536461 12 537229 12 537997 12 538765 12 539533 12 540301 12 541069 12 541837 12 542605 12 543373 12 544141 12 544909 12 545677 12 546445 12 547213 12 547981 12 548749 12 549517 12 550285 12 551053 12 551821 12 552589 12 553357 12 554125 12 554893 12 555661 12 556429 12 557197 12 557965 12 558733 12 559501 12 560269 12 561037 12 561805 12 562573 12 563341 12 564109 12 564877 12 565645 12 566413 12 567181 12 567949 12 568717 12 569485 12 570253 12 571021 12 571789 12 572557 12 573325 12 574093 12 574861 12 575629 12 576397 12 577165 12 577933 12 578701 12 579469 12 580237 12 581005 12 581773 12 582541 12 583309 12 584077 12 584845 12 585613 12 586381 12 587149 12 587917 12 588685 12 589453 12 590221 12 590989 12 591757 12 592525 12 593293 12 594061 12 594829 12 595597 12 596365 12 597133 12 597901 12 598669 12 599437 12 600205 12 600973 12 601741 12 602509 12 603277 12 604045 12 604813 12 605581 12 606349 12 607117 12 607885 12 608653 12 609421 12 610189 12 610957 12 611725 12 612493 12 613261 12 614029 12 614797 12 615565 12 616333 12 617101 12 617869 12 618637 12 619405 12 620173 12 620941 12 621709 12 622477 12 623245 12 624013 12 624781 12 625549 12 626317 12 627085 12 627853 12 628621 12 629389 12 630157 12 630925 12 631693 12 632461 12 633229 12 633997 12 634765 12 635533 12 636301 12 637069 12 637837 12 638605 12 639373 12 640141 12 640909 12 641677 12 642445 12 643213 12 643981 12 644749 12 645517 12 646285 12 647053 12 647821 12 648589 12 649357 12 650125 12 650893 12 651661 12 652429 12 653197 12 653965 12 654733 12 655501 12 656269 12 657037 12 657805 12 658573 12 659341 12 660109 12 660877 12 661645 12 662413 12 663181 12 663949 12 664717 12 665485 12 666253 12 667021 12 667789 12 668557 12 669325 12 670093 12 670861 12 671629 12 672397 12 673165 12 673933 12 674701 12 675469 12 676237 12 677005 12 677773 12 678541 12 679309 12 680077 12 680845 12 681613 12 682381 12 683149 12 683917 12 684685 12 685453 12 686221 12 686989 12 687757 12 688525 12 689293 12 690061 12 690829 12 691597 12 692365 12 693133 12 693901 12 694669 12 695437 12 696205 12 696973 12 697741 12 698509 12 699277 12 700045 12 700813 12 701581 12 702349 12 703117 12 703885 12 704653 12 705421 12 706189 12 706957 12 707725 12 708493 12 709261 12 710029 12 710797 12 711565 12 712333 12 713101 12 713869 12 714637 12 715405 12 716173 12 716941 12 717709 12 718477 12 719245 12 720013 12 720781 12 721549 12 722317 12 723085 12 723853 12 724621 12 725389 12 726157 12 726925 12 727693 12 728461 12 729229 12 730000 12 730760 12 731520 12 732280 12 733040 12 733800 12 734560 12 735320 12 736080 12 736840 12 737600 12 738360 12 739120 12 739880 12 740640 12 741400 12 742160 12 742920 12 743680 12 744440 12 745200 12 745960 12 746720 12 747480 12 748240 12 749000 12 749760 12 750520 12 751280 12 752040 12 752800 12 753560 12 754320 12 755080 12 755840 12 756600 12 757360 12 758120 12 758880 12 759640 12 760400 12 761160 12 761920 12 762680 12 763440 12 764200 12 764960 12 765720 12 766480 12 767240 12 768000 12 768760 12 769520 12 770280 12 771040 12 771800 12 772560 12 773320 12 774080 12 774840 12 775600 12 776360 12 777120 12 777880 12 778640 12 779400 12 780160 12 780920 12 781680 12 782440 12 783200 12 783960 12 784720 12 785480 12 786240 12 787000 12 787760 12 788520 12 789280 12 790040 12 790800 12 791560 12 792320 12 793080 12 793840 12 794600 12 795360 12 796120 12 796880 12 797640 12 798400 12 799160 12 799920 12 800680 12 801440 12 802200 12 802960 12 803720 12 804480 12 805240 12 806000 12 806760 12 807520 12 808280 12 809040 12 809800 12 810560 12 811320 12 812080 12 812840 12 813600 12 814360 12 815120 12 815880 12 816640 12 817400 12 818160 12 818920 12 819680 12 820440 12 821200 12 821960 12 822720 12 823480 12 824240 12 825000 12 825760 12 826520 12 827280 12 828040 12 828800 12 829560 12 830320 12 831080 12 831840 12 832600 12 833360 12 834120 12 834880 12 835640 12 836400 12 837160 12 837920 12 838680 12 839440 12 840200 12 840960 12 841720 12 842480 12 843240 12 844000 12 844760 12 845520 12 846280 12 847040 12 847800 12 848560 12 849320 12 850080 12 850840 12 851600 12 852360 12 853120 12 853880 12 854640 12 855400 12 856160 12 856920 12 857680 12 858440 12 859200 12 859960 12 860720 12 861480 12 862240 12 863000 12 863760 12 864520 12 865280 12 866040 12 866800 12 867560 12 868320 12 869080 12 869840 12 870600 12 871360 12 872120 12 872880 12 873640 12 874400 12 875160 12 875920 12 876680 12 877440 12 878200 12 878960 12 879720 12 880480 12 881240 12 882000 12 882760 12 883520 12 884280 12 885040 12 885800 12 886560 12 887320 12 888080 12 888840 12 889600 12 890360 12 891120 12 891880 12 892640 12 893400 12 894160 12 894920 12 895680 12 896440 12 897200 12 897960 12 898720 12 899480 12 900240 12 901000 12 901760 12 902520 12 903280 12 904040 12 904800 12 905560 12 906320 12 907080 12 907840 12 908600 12 909360 12 910120 12 910880 12 911640 12 912400 12 913160 12 913920 12 914680 12 915440 12 916200 12 916960 12 917720 12 918480 12 919240 12 920000 12 920760 12 921520 12 922280 12 923040 12 923800 12 924560 12 925320 12 926080 12 926840 12 927600 12 928360 12 929120 12 929880 12 930640 12 931400 12 932160 12 932920 12 933680 12 934440 12 935200 12 935960 12 936720 12 937480 12 938240 12 939000 12 939760 12 940520 12 941280 12 942040 12 942800 12 943560 12 944320 12 945080 12 945840 12 946600 12 947360 12 948120 12 948880 12 949640 12 950400 12 951160 12 951920 12 952680 12 953440 12 954200 12 954960 12 955720 12 956480 12 957240 12 958000 12 958760 12 959520 12 960280 12 961040 12 961800 12 962560 12 963320 12 964080 12 964840 12 965600 12 966360 12 967120 12 967880 12 968640 12 969400 12 970160 12 970920 12 971680 12 972440 12 973200 12 973960 12 974720 12 975480 12 976240 12 977000 12 977760 12 978520 12 979280 12 980040 12 980800 12 981560 12 982320 12 983080 12 983840 12 984600 12 985360 12 986120 12 986880 12 987640 12 988400 12 989160 12 989920 12 990680 12 991440 12 992200 12 992960 12 993720 12 994480 12 995240 12 996000 12 996760 12 997520 12 998280 12 999040 12 999800 12 1000000']
27 ground_Truth = decodeGroundTruth(en_pix2, img1.shape, image1)
28 kmeans_mask1 = kmeans_segmentator(img2)
29 en_pix1 = ['308744 1 309511 4 310279 6 311046 9 311814 11 312581 14 313349 16 314116 19 314884 21 315651 24 316419 26 317187 28 317954 30 318722 32 319489 34 320257 36 321025 38 321793 40 322560 42 323328 44 324096 46 324864 48 325632 50 326400 52 327168 54 327936 56 328704 58 329472 60 330240 62 331008 64 331776 66 332544 68 333312 70 334080 72 334848 74 335616 76 336384 78 337152 80 337920 82 338688 84 339456 86 340224 88 340992 90 341760 92 342528 94 343296 96 344064 98 344832 100 345600 102 346368 104 347136 106 347904 108 348672 110 349440 112 350208 114 350976 116 351744 118 352512 120 353280 122 354048 124 354816 126 355584 128 356352 130 357120 132 357888 134 358656 136 359424 138 360192 140 360960 142 361728 144 362496 146 363264 148 364032 150 364800 152 365568 154 366336 156 367104 158 367872 160 368640 162 369408 164 370176 166 370944 168 371712 170 372480 172 373248 174 374016 176 374784 178 375552 180 376320 182 377088 184 377856 186 378624 188 379392 190 380160 192 380928 194 381696 196 382464 198 383232 200 384000 202 384768 204 385536 206 386304 208 387072 210 387840 212 388608 214 389376 216 390144 218 390912 220 391680 222 392448 224 393216 226 393984 228 394752 230 395520 232 396288 234 397056 236 397824 238 398592 240 399360 242 400128 244 400896 246 401664 248 402432 250 403200 252 403968 254 404736 256 405504 258 406272 260 407040 262 407808 264 408576 266 409344 268 410112 270 410880 272 411648 274 412416 276 413184 278 413952 280 414720 282 415488 284 416256 286 417024 288 417792 290 418560 292 419328 294 420096 296 420864 298 421632 300 422400 302 423168 304 423936 306 424704 308 425472 310 426240 312 427008 314 427776 316 428544 318 429312 320 430080 322 430848 324 431616 326 432384 328 433152 330 433920 332 434688 334 435456 336 436224 338 436992 340 437760 342 438528 344 439296 346 440064 348 440832 350 441600 352 442368 354 443136 356 443904 358 444672 360 445440 362 446208 364 446976 366 447744 368 448512 370 449280 372 450048 374 450816 376 451584 378 452352 380 453120 382 453888 384 454656 386 455424 388 456192 390 456960 392 457728 394 458496 396 459264 398 460032 400 460800 402 461568 404 462336 406 463104 408 463872 410 464640 412 465408 414 466176 416 466944 418 467712 420 468480 422 469248 424 470016 426 470784 428 471552 430 472320 432 473088 434 473856 436 474624 438 475392 440 476160 442 476928 444 477696 446 478464 448 479232 450 480000 452 480768 454 481536 456 482304 458 483072 460 483840 462 484608 464 485376 466 486144 468 486912 470 487680 472 488448 474 489216 476 490000 478 490760 480 491520 482 492280 484 493040 486 493800 488 494560 490 495320 492 496080 494 496840 496 497600 498 498360 500 499120 502 499880 504 500640 506 501400 508 502160 510 502920 512 503680 514 504440 516 505200 518 505960 520 506720 522 507480 524 508240 526 509000 528 509760 530 510520 532 511280 534 512040 536 512800 538 513560 540 514320 542 515080 544 515840 546 516600 548 517360 550 518120 552 518880 554 519640 556 520400 558 521160 560 521920 562 522680 564 523440 566 524200 568 524960 570 525720 572 526480 574 527240 576 528000 578 528760 580 529520 582 530280 584 531040 586 531800 588 532560 590 533320 592 534080 594 534840 596 535600 598 536360 600 537120 602 537880 604 538640 606 539400 608 540160 610 540920 612 541680 614 542440 616 543200 618 543960 620 544720 622 545480 624 546240 626 547000 628 547760 630 548520 632 549280 634 550040 636 550800 638 551560 640 552320 642 553080 644 553840 646 554600 648 555360 650 556120 652 556880 654 557640 656 558400 658 559160 660 559920 662 560680 664 561440 666 562200 668 562960 670 563720 672 564480 674 565240 676 566000 678 566760 680 567520 682 568280 684 569040 686 569800 688 570560 690 571320 692 572080 694 572840 696 573600 698 574360 700 575120 702 575880 704 576640 706 577400 708 578160 710 578920 712 579680 714 580440 716 581200 718 581960 720 582720 722 583480 724 584240 726 585000 728 585760 730 586520 732 587280 734 588040 736 588800 738 589560 740 590320 742 591080 744 591840 746 592600 748 593360 750 594120 752 594880 754 595640 756 596400 758 597160 760 597920 762 598680 764 599440 766 600200 768 600960 770 601720 772 602480 774 603240 776 604000 778 604760 780 605520 782 606280 784 607040 786 607800 788 608560 790 609320 792 610080 794 610840 796 611600 798 612360 800 613120 802 613880 804 614640 806 615400 808 616160 810 616920 812 617680 814 618440 816 619200 818 619960 820 620720 822 621480 824 622240 826 623000 828 623760 830 624520 832 625280 834 626040 836 626800 838 627560 840 628320 842 629080 844 629840 846 630600 848 631360 850 632120 852 632880 854 633640 856 634400 858 635160 860 635920 862 636680 864 637440 866 638200 868 638960 870 639720 872 640480 874 641240 876 642000 878 642760 880 643520 882 644280 884 645040 886 645800 888 646560 890 647320 892 648080 894 648840 896 649600 898 650360 900 651120 902 651880 904 652640 906 653400 908 654160 910 654920 912 655680 914 656440 916 657200 918 657960 920 658720 922 659480 924 660240 926 661000 928 661760 930 662520 932 663280 934 664040 936 664800 938 665560 940 666320 942 667080 944 667840 946 668600 948 669360 950 670120 952 670880 954 671640 956 672400 958 673160 960 673920 962 674680 964 675440 966 676200 968 676960 970 677720 972 678480 974 679240 976 680000 978 680760 980 681520 982 682280 984 683040 986 683800 988 684560 990 685320 992 686080 994 686840 996 687600 998 688360 1000 689120']
30 ground_Truth1 = decodeGroundTruth(en_pix1, img2.shape, image2)
31
32 def mostrar(imagen, mask1, mask2, titulo_segmentador):
33     fig, (ax1,ax2,ax3) = plt.subplots(1,3)
34     fig.set_size_inches(10,10)
35     ax1.imshow(imagen)
36     ax1.set_title("Imagen Original")
37     ax1.axis('off')
38     ax2.imshow(mask1, cmap='gray')
39     ax2.set_title("Ground Truth")
40     ax2.axis('off')
41     ax3.imshow(mask2, cmap='gray')
42     ax3.set_title(titulo_segmentador)
43     ax3.axis('off')
44     plt.show();
45
46 mostrar(img1, ground_Truth, kmeans_mask, "Segmentador K means
```

Promedio de las métricas:  
TP: 1857.1857142857143 FP: 104277.57142857143  
TN: 480378.9285714286 FN: 3310.3142857142857  
Especificidad: 0.8216580200813163 Precision: 0.41391692981282135 Recall: 0.4090340873861649

## ▼ Segmentador Canny

El segmentador Canny es un segmentador bastante simple y computacionalmente menos costoso que los otros dos segmentadores propuestos. Inicialmente hace un suavizado de la imagen y, sobre la imagen suavizada, detecta los bordes utilizando Canny. Finalmente, se aplica una dilatación para rellenar los objetos, seguida de una erosión para reducir los contornos.

```
1 def canny_segmentator(img):
2     blur = cv2.medianBlur(np.array(img, dtype = 'uint8'), 3)
3     img1_gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY) # Imagen en blanco y negro
4     canny = cv2.Canny(img1_gray, 60, 250) # Detección de bordes con Canny
5     img1_dilat = dilation(canny, disk(6)) # Dilatación
6     img1_erod = erosion(img1_dilat, disk(2)) # Erosión
7     return np.where(img1_erod == 0, 0, 255)/255 # Se retorna la máscara
8
9 canny_mask = canny_segmentator(img1)
10 canny_mask2 = canny_segmentator(img2)
11
12 mostrar(img1, ground_Truth, canny_mask, "Segmentador Canny")
13 mostrar(img2, ground_Truth1, canny_mask2, "Segmentador Canny")
```



Se evalúa el segmentador generado:

```
1 test_segmentator(canny_segmentator)
```

Promedio de las métricas:  
TP: 3834.157142857143 FP: 85466.4  
TN: 499190.1 FN: 1333.3428571428572  
Especificidad: 0.853640121980766 Precision: 0.4932145242967803 Recall: 0.7687998514158321

## ▼ Segmentador felzenszwalb

Este segmentador produce una sobresegmentación de la imagen usando un rápido y mínimo agrupamiento basado en árboles de expansión. En cuanto a los parámetros de la función, la magnitud de scale determina la configuración de los segmentos, es decir, un valor alto de esta variable implica que el número de segmentos será menor pero estos tendrán un tamaño mayor. Por otro lado, sigma indica el tamaño del kernel Gaussiano que se utiliza para suavizar la imagen de entrada, y min\_size determina el tamaño mínimo que pueden alcanzar las regiones. Los valores de estos parámetros se han obtenido de forma empírica, comprobando que se adaptan correctamente a la mayoría de las imágenes escogidas.

```
1 def felzenszwalb_segmentator(img):
2     segments_fz = felzenszwalb(img, scale=400, sigma=4, min_size=400)
3     img_fel_1 = color.label2rgb(segments_fz, img, kind='avg', bg_label=0, bg_color=0)
4     img_fel_1[img_fel_1 < 0.1] = 255
```

```

4 img_fel1[img_fel1 > 0] = 255
5 temp = np.where(img_fel1 == 0, 0, 255)/255
6 new = np.zeros((temp.shape[0],temp.shape[1]))
7 for i in range(temp.shape[0]): #Se eliminan las capas que no nos interesan
8     for j in range(temp.shape[1]):
9         new[i][j] = temp[i][j][0]
10 return new
11
12 felzenszwalb_mask = felzenszwalb_segmentator(img1)
13 felzenszwalb_mask2 = felzenszwalb_segmentator(img2)
14
15 mostrar(img1, ground_Truth, felzenszwalb_mask, "Segmentador Felzenszwalb")
16 mostrar(img2, ground_Truth1, felzenszwalb_mask2, "Segmentador Felzenszwalb")

```



Se evalúa el segmentador generado:

```

1 test_segmentator(felzenszwalb_segmentator)

```

Promedio de las métricas:  
TP: 5041.142857142857 FP: 21405.64285714286  
TN: 563250.8571428572 FN: 126.35714285714286  
Especificidad: 0.9635313199131909 Precision: 0.4669846752219528 Recall: 0.9678265276759055

## ▼ Segmentador Agregado

Este segmentador funciona agregando los resultados de las máscaras binarias obtenidas por otros segmentadores. El método por el que se realiza esta agregación es por medio de un sistema de votación en el que hay 3 posibles situaciones:

- Hay un empate en la votación de los segmentadores y por tanto se elige la selección del mejor
- Consenso en votación positiva
- Consenso en votación negativa

Esta agregación genera, como los otros segmentadores, una máscara binaria.

```

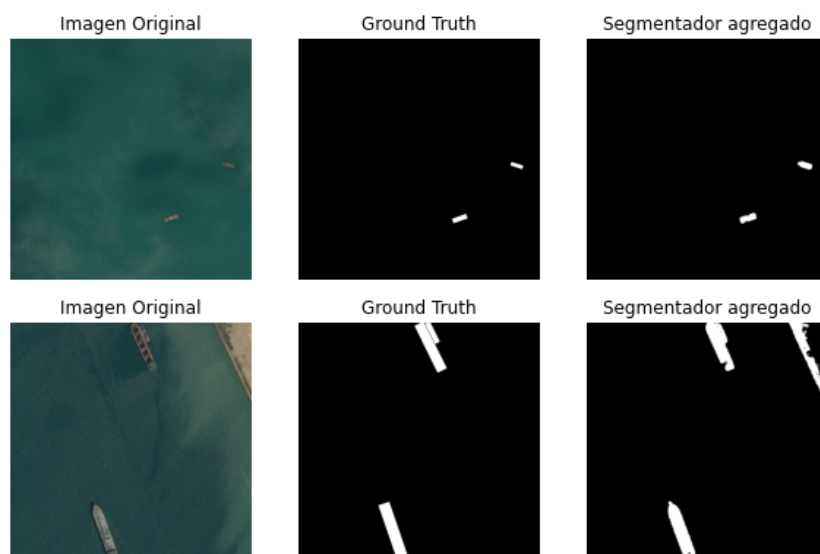
1 def segmenter_agregado(image):
2     n, m, dummy = image.shape
3     seg1_mask = kmeans_segmentator(image)
4     seg2_mask = canny_segmentator(image)
5     seg3_mask = felzenszwalb_segmentator(image)
6     output = np.zeros((n,m))
7     for i in range(n):
8         for j in range(m):
9             mask_sum = seg1_mask[i][j] + seg2_mask[i][j] + seg3_mask[i][j]
10            if mask_sum == 1.5:
11                output[i][j] = seg1_mask[i][j] # Si hay mismo número de barcos que no barcos se selecciona el mejor segmer
12            elif mask_sum < 1.5:
13                output[i][j] = 0 # Si hay menos de 2 que seleccionen barco es mar
14            else:
15                output[i][j] = 1 #Si hay mas de 2 que son barco es barco
16 return output

```

```

16 return output
17
18 agregado_mask = segmerter_agregado( img1)
19 agregado_mask2 = segmerter_agregado( img2)
20 mostrar(img1, ground_Truth, agregado_mask, "Segmentador agregado")
21 mostrar(img2, ground_Truth1, agregado_mask2, "Segmentador agregado")

```



Se evalúa el segmentador generado:

```

1 test_segmentator(segmerter_agregado)

```

```

Promedio de las métricas:
TP: 4147.1  FP: 65460.0
TN: 519196.5  FN: 1020.4
Especificidad: 0.8879256033321499  Precision: 0.5472948549210893  Recall: 0.8428666390778291

```

## ▼ Resultados

A continuación se muestran los promedios de las métricas calculadas para los diferentes segmentadores desarrollados:

	Kmeans	Canny	Felzenszwalb	Agregado
Especificidad	0.8216	0.8536	0.9635	0.8879
Precisión	0.4139	0.4932	0.4669	0.5472
Recall	0.4090	0.7687	0.9678	0.8428

*\*Si se desea visualizar todas las imágenes del dataset y su resultado se debe modificar la variable "debug" en la función test\_segmentador*

## Conclusiones

Como se puede observar el segmentador agregado mejora la precisión a costa de una menor especificidad y Recall que el mejor segmentador obtenido (Felzenszwalb). El agregado se muestra ligeramente más robusto frente a imágenes con mucho ruido, ya sea de nubes o de oleaje.

El hecho de que se hayan usado los mismos parámetros de entrada para todas las imágenes hace que los resultados obtenidos empeoren, pero debido a la complejidad de automatizar la selección de parámetros para cada imagen se ha optado por no implementarlo.

También cabe destacar que el ground truth de este dataset no se ajusta a la silueta de los barcos a identificar, sino que señala el área rectangular donde estos se encuentran, empeorando así las métricas de los segmentadores puesto que las máscaras generadas si se ajustan a esta silueta.

## 7. Bibliografía

[1] 19/02/2021, <https://medium.com/analytics-vidhya/generating-masks-from-encoded-pixels-semantic-segmentation-18635e834ad0>

[2] 19/02/2021, <https://www.kaggle.com/c/airbus-ship-detection/overview>