|  |
| --- |
| import cv2 import numpy as np from sklearn.linear\_model import LinearRegression  # 그레이스케일 이미지 불러오기 original\_image = cv2.imread('difficult2.png', cv2.IMREAD\_GRAYSCALE)  # 큰 ROI 설정 x, y, w, h = 30, 30, 200, 370 # difficult2  # ROI 시각적 표시를 위한 컬러 변환 image\_with\_roi = cv2.cvtColor(original\_image, cv2.COLOR\_GRAY2BGR)  # ROI 강조 표시 cv2.rectangle(image\_with\_roi, (x, y), (x + w, y + h), (0, 255, 0), 2)  # 원본 이미지 전체에 밝기 값 45 이하를 0으로 처리 threshold\_value = 45 \_, thresholded\_image = cv2.threshold(original\_image, threshold\_value, 255, cv2.THRESH\_TOZERO)  # Non-Local Means 필터 적용 nlm\_denoised = cv2.fastNlMeansDenoising(  thresholded\_image, None, h=10, templateWindowSize=7, searchWindowSize=21 )  # 첫 번째와 두 번째로 찍힌 점들의 리스트 초기화 first\_points = [] second\_points = []  # ROI 내부에서 밝기 변화 감지하여 점 찍기 for i in range(y + 1, y + h - 1): # ROI 내부의 y 좌표 (위에서 아래로)  points\_marked = 0 # 매 행(row)마다 points\_marked를 초기화  previous\_value = None # 각 행(row)마다 previous\_value 초기화  last\_marked\_position = None   for j in range(w - 1, -1, -1): # ROI 내부의 x 좌표 (오른쪽에서 왼쪽으로)  pixel\_value = int(thresholded\_image[i, x + j]) # 현재 (x + j, i) 위치의 픽셀 값   if previous\_value is not None:  delta = abs(pixel\_value - previous\_value) # 이전 픽셀과의 밝기 차이 계산  if delta >= 5:  if points\_marked == 0 or (points\_marked < 2 and abs(j - last\_marked\_position) > 5):  # 변화 감지 지점에 빨간 점 표시  cv2.circle(image\_with\_roi, (x + j, i), 1, (0, 0, 255), -1)   # 첫 번째와 두 번째 점에 따라 리스트에 추가  if points\_marked == 0:  first\_points.append((x + j, i)) # 첫 번째 점 리스트에 추가  elif points\_marked == 1:  second\_points.append((x + j, i)) # 두 번째 점 리스트에 추가   points\_marked += 1 # 변화 지점 카운트 증가  last\_marked\_position = j   # 두 개의 변화 지점을 찾았으면 더 이상 점 찍기 중지  if points\_marked == 2:  break   previous\_value = pixel\_value # 이전 픽셀 값 업데이트   # 결과 시각화 cv2.imshow('ROI Visualization', image\_with\_roi) cv2.waitKey(0) cv2.destroyAllWindows() |

실행결과

스크린샷, 텍스트, 멀티미디어 소프트웨어이(가) 표시된 사진

자동 생성된 설명

|  |
| --- |
| 첫 번째와 두 번째 점의 기준 좌표를 구하는 데 있어, 각각의 좌표를 대표할 수 있는 방법을 선택하는 것은 목표에 따라 달라질 수 있습니다. 각 방법의 장단점을 고려하여 어떤 경우에 적합한지 설명하겠습니다.  **1. 평균값 (Mean)**   * **설명**: 각 점들의 좌표의 평균값을 구하여 기준 좌표로 사용하는 방식입니다. * **장점**: 데이터가 정상적으로 분포되어 있으면 모든 점을 고려하기 때문에, 편향 없이 전체적인 중심을 잘 나타낼 수 있습니다. * **단점**: 아웃라이어(이상치)에 영향을 많이 받습니다. 특정 좌표가 극단적으로 크거나 작다면 전체 평균값이 왜곡될 수 있습니다. * **적합한 경우**: 점들이 대체로 균일하게 분포되어 있고, 이상치가 많지 않은 경우.   **2. 중앙값 (Median)**   * **설명**: 각 점의 x좌표와 y좌표 각각의 중앙값을 사용하여 기준 좌표를 설정합니다. * **장점**: 아웃라이어에 덜 민감합니다. 이상치가 있더라도 중앙값은 큰 영향을 받지 않습니다. * **단점**: 대칭적인 분포가 아닌 경우, 중앙값이 데이터의 중심을 항상 잘 나타내지는 않을 수 있습니다. * **적합한 경우**: 이상치가 존재하거나, 점들이 고르게 분포되어 있지 않은 경우.   **3. 선형 회귀 (Linear Regression)**   * **설명**: 첫 번째와 두 번째 점들이 어떤 패턴을 형성한다고 가정할 때, 해당 패턴을 직선으로 근사하여 그 중심을 대표 좌표로 설정합니다. * **장점**: 점들이 특정 방향으로 정렬되었거나 직선 형태로 나타나는 경우 효과적입니다. * **단점**: 점들이 산발적으로 분포된 경우, 선형 회귀를 사용하면 대표 좌표가 실제 중심에서 벗어날 수 있습니다. * **적합한 경우**: 점들이 일정한 방향으로 패턴을 가지고 있는 경우(예: 경계선 상의 점들).   **4. 최빈값 (Mode)**   * **설명**: 가장 빈번하게 나타나는 좌표 값을 대표로 사용합니다. * **장점**: 특정 좌표에 점이 집중되어 있을 때, 이를 기준으로 삼기 쉽습니다. * **단점**: 점들의 분포가 고르지 않은 경우, 실제 중심을 잘 나타내지 못할 수 있습니다. 모든 좌표가 고유하면 최빈값을 구하기 어려울 수 있습니다. * **적합한 경우**: 점들이 몇몇 특정 위치에 집중되어 있을 때.   **권장 사항**   * **데이터가 특정 패턴 없이 고르게 분포**되어 있다면, **평균값**이나 **중앙값**이 좋은 선택입니다. * **이상치가 많은 경우**라면, **중앙값**이 더 안정적인 기준 좌표가 될 것입니다. * **점들이 경계선을 따라 특정 방향으로 정렬된 경우**라면, **선형 회귀**를 통해 경향을 파악한 후 중심을 구하는 것이 좋습니다. * **특정 위치에 점이 많이 집중된 경우**라면, **최빈값**을 사용해 해당 좌표를 대표 좌표로 삼을 수 있습니다. |

|  |
| --- |
| 각 방법의 특성과 현재 점의 분포를 고려할 때, **중앙값**이나 **선형 회귀**가 적합해 보입니다. 이유는 다음과 같습니다:   1. **중앙값**:    * **이상치**(outlier)에 강하고, 분포가 일정하지 않더라도 대표성을 잘 유지합니다.    * 점들이 ROI 경계에 따라 줄지어 찍혀있는 경우, 특정 방향으로 분포가 있어도 중앙값은 중심 좌표를 잘 반영할 가능성이 큽니다. 2. **선형 회귀**:    * 빨간 점들이 경계선이나 특정 패턴에 따라 연속적으로 찍혀 있는 경우, 선형 회귀를 사용해 이들의 경향성을 반영할 수 있습니다.    * 이 경우, 선형 회귀를 통해 각 좌표에 대한 회귀선을 만들고, 이 선의 중심을 기준 좌표로 잡는 방식이 유리할 수 있습니다.   **방법 선택 기준**   * **중앙값**은 간단하게 점의 중심을 찾는 데 적합하며, 이상치에 덜 민감합니다. * **선형 회귀**는 경계선을 따라 점들이 일직선이나 특정 경향을 띄고 있을 때 적합합니다. 현재 빨간 점들이 경계선에 줄지어 있다면, 선형 회귀가 더 의미 있는 결과를 줄 수 있습니다. |

|  |
| --- |
| import cv2 import numpy as np from sklearn.linear\_model import LinearRegression  # 그레이스케일 이미지 불러오기 original\_image = cv2.imread('difficult2.png', cv2.IMREAD\_GRAYSCALE)  # 큰 ROI 설정 x, y, w, h = 30, 30, 200, 370 # difficult2  # ROI 시각적 표시를 위한 컬러 변환 image\_with\_roi = cv2.cvtColor(original\_image, cv2.COLOR\_GRAY2BGR)  # ROI 강조 표시 cv2.rectangle(image\_with\_roi, (x, y), (x + w, y + h), (0, 255, 0), 2)  # 원본 이미지 전체에 밝기 값 45 이하를 0으로 처리 threshold\_value = 45 \_, thresholded\_image = cv2.threshold(original\_image, threshold\_value, 255, cv2.THRESH\_TOZERO)  # Non-Local Means 필터 적용 nlm\_denoised = cv2.fastNlMeansDenoising(  thresholded\_image, None, h=10, templateWindowSize=7, searchWindowSize=21 )  # 첫 번째와 두 번째로 찍힌 점들의 리스트 초기화 first\_points = [] second\_points = []  # ROI 내부에서 밝기 변화 감지하여 점 찍기 for i in range(y + 1, y + h - 1): # ROI 내부의 y 좌표 (위에서 아래로)  points\_marked = 0 # 매 행(row)마다 points\_marked를 초기화  previous\_value = None # 각 행(row)마다 previous\_value 초기화  last\_marked\_position = None   for j in range(w - 1, -1, -1): # ROI 내부의 x 좌표 (오른쪽에서 왼쪽으로)  pixel\_value = int(thresholded\_image[i, x + j]) # 현재 (x + j, i) 위치의 픽셀 값   if previous\_value is not None:  delta = abs(pixel\_value - previous\_value) # 이전 픽셀과의 밝기 차이 계산  if delta >= 5:  if points\_marked == 0 or (points\_marked < 2 and abs(j - last\_marked\_position) > 5):  # 변화 감지 지점에 빨간 점 표시  cv2.circle(image\_with\_roi, (x + j, i), 1, (0, 0, 255), -1)   # 첫 번째와 두 번째 점에 따라 리스트에 추가  if points\_marked == 0:  first\_points.append((x + j, i)) # 첫 번째 점 리스트에 추가  elif points\_marked == 1:  second\_points.append((x + j, i)) # 두 번째 점 리스트에 추가   points\_marked += 1 # 변화 지점 카운트 증가  last\_marked\_position = j   # 두 개의 변화 지점을 찾았으면 더 이상 점 찍기 중지  if points\_marked == 2:  break   previous\_value = pixel\_value # 이전 픽셀 값 업데이트  # 각 행마다 첫 번째 점들과 두 번째 점들의 리스트 출력 print("첫 번째 점들의 리스트:", first\_points) print("두 번째 점들의 리스트:", second\_points)  # 첫 번째 점들의 x, y 좌표 추출 first\_x = [point[0] for point in first\_points] first\_y = [point[1] for point in first\_points]  # 두 번째 점들의 x, y 좌표 추출 second\_x = [point[0] for point in second\_points] second\_y = [point[1] for point in second\_points]  # 중앙값 계산 first\_median = (np.median(first\_x), np.median(first\_y)) second\_median = (np.median(second\_x), np.median(second\_y))  # 선형 회귀 계산 # 첫 번째 점들에 대한 선형 회귀 first\_x\_np = np.array(first\_x).reshape(-1, 1) first\_y\_np = np.array(first\_y) first\_regressor = LinearRegression() first\_regressor.fit(first\_x\_np, first\_y\_np) first\_line\_center\_x = np.mean(first\_x) first\_line\_center\_y = first\_regressor.predict(np.array([[first\_line\_center\_x]]))[0]  # 두 번째 점들에 대한 선형 회귀 second\_x\_np = np.array(second\_x).reshape(-1, 1) second\_y\_np = np.array(second\_y) second\_regressor = LinearRegression() second\_regressor.fit(second\_x\_np, second\_y\_np) second\_line\_center\_x = np.mean(second\_x) second\_line\_center\_y = second\_regressor.predict(np.array([[second\_line\_center\_x]]))[0]  # 결과 출력 print("첫 번째 점들의 중앙값:", first\_median) print("두 번째 점들의 중앙값:", second\_median) print("첫 번째 점들의 선형 회귀 중심:", (first\_line\_center\_x, first\_line\_center\_y)) print("두 번째 점들의 선형 회귀 중심:", (second\_line\_center\_x, second\_line\_center\_y))  # 결과 시각화 cv2.imshow('ROI Visualization', image\_with\_roi) cv2.waitKey(0) cv2.destroyAllWindows() |

결과

|  |
| --- |
| 첫 번째 점들의 중앙값: (np.float64(207.0), np.float64(214.5))  두 번째 점들의 중앙값: (np.float64(142.0), np.float64(214.5))  첫 번째 점들의 선형 회귀 중심: (np.float64(205.27445652173913), np.float64(214.5))  두 번째 점들의 선형 회귀 중심: (np.float64(143.0516304347826), np.float64(214.5)) |

|  |
| --- |
| 출력된 결과를 보면, 중앙값과 선형 회귀 중심이 각각 비슷한 값을 보이고 있으며, **y 좌표**는 두 경우 모두 **214.5로 동일**합니다. 이는 빨간 점들이 수직 방향으로 일관되게 정렬되어 있음을 시사합니다.  **결과 해석 및 추천 기준**   1. **중앙값**:    * 중앙값이 이상치에 강하고 전체 데이터의 중심을 잘 나타내는 특성을 가지고 있으므로, 각 행의 중심을 찾는 데 적합합니다.    * 결과에서 (207.0, 214.5)와 (142.0, 214.5)로, 점의 분포 중심을 비교적 잘 나타냅니다. 2. **선형 회귀 중심**:    * 선형 회귀 중심은 각각 (205.27, 214.5)와 (143.05, 214.5)로 계산되어 있습니다. 이는 각 점들이 거의 직선으로 분포되어 있다는 의미입니다.    * 점들이 수직 방향으로 정렬되어 있는 경우 선형 회귀를 통한 중심 계산이 중앙값과 크게 차이 나지 않으므로, 선형 회귀가 데이터를 복잡하게 처리할 필요가 없는 단순 수직 분포에서는 추가적인 이점을 제공하지 않을 수 있습니다.   **결론 및 추천**  **중앙값**을 사용하여 (207.0, 214.5)와 (142.0, 214.5)를 기준 좌표로 사용하는 것이 좋습니다. 중앙값이 계산도 간단하고, 이 경우 선형 회귀와 큰 차이가 없으므로 추가적인 계산 부담 없이 적합한 결과를 얻을 수 있습니다.  중앙값이 최종 기준으로 결정되었다면, 코드에서 선형 회귀 부분을 제거하고 중앙값을 기준으로 설정하시면 됩니다. |

**방법 비교**

| **방법** | **장점** | **단점** | **추천 상황** |
| --- | --- | --- | --- |
| **KMeans 2번** | 직관적이며 군집 수가 명확할 때 효과적 | 군집 수가 적합하지 않을 경우 부정확할 수 있음 | 군집 수가 두 그룹으로 나뉘는 경우 |
| **DBSCAN** | 군집 수를 사전 설정하지 않음 | eps와 min\_samples 설정에 민감 | 데이터 분포가 불균등할 때 적합 |
| **히스토그램 기반 필터링** | 간단하고 빠름 | 군집화보다 정밀도가 떨어질 수 있음 | 대략적인 y값 탐지가 필요한 경우 |

|  |
| --- |
| **eps와 min\_samples에 대한 설명**  **1. eps (군집화 반경)**   * eps는 DBSCAN에서 **포인트 간 거리**를 나타냅니다. * 하나의 데이터 포인트가 군집의 일부로 간주되려면, 해당 포인트를 중심으로 한 eps 반경 내에 min\_samples 개 이상의 데이터 포인트가 존재해야 합니다. * **값의 의미**:   + eps가 **작을수록** 데이터가 더 조밀해야 군집이 형성됩니다.   + eps가 **클수록** 더 넓은 영역에서 포인트들을 군집으로 묶습니다.   **2. min\_samples (최소 샘플 수)**   * min\_samples는 군집을 구성하기 위해 **eps 반경 내에 포함되어야 하는 최소 데이터 포인트 수**를 설정합니다. * 군집을 이루는 데이터의 밀도 기준을 결정합니다. * **값의 의미**:   + 값이 **작을수록** 군집이 더 쉽게 형성됩니다.   + 값이 **클수록** 군집이 형성되기 더 어려워지고, 더 강한 밀도를 요구합니다.   **왜 eps=3과 min\_samples=5로 설정했는가?**  **1. eps=3**   * 좌표 데이터는 (x, y)로 구성되어 있으며, 데이터 간 거리의 단위는 Euclidean distance(유클리드 거리)입니다. * eps=3은 데이터 포인트 간 거리가 3 이하일 때 군집으로 묶이도록 설정합니다. * **직관적인 이유**:   + 주어진 좌표 데이터를 보면 많은 포인트가 비슷한 범위에 몰려 있습니다. 거리 3을 반경으로 설정하면 이 밀집된 데이터들을 적절히 군집으로 묶을 수 있습니다.   **2. min\_samples=5**   * 군집이 형성되려면, 반경 eps=3 내에 적어도 **5개의 데이터 포인트**가 있어야 합니다. * **값을 설정한 이유**:   + 데이터가 비교적 촘촘히 분포된 경우, 5개 정도의 포인트를 기준으로 하면 노이즈를 걸러내면서도 실제 군집을 탐지할 수 있습니다.   + 더 큰 값을 설정하면 데이터 밀도가 더 높아야 군집으로 인식되므로 군집이 잘 형성되지 않을 수 있습니다.   **eps와 min\_samples 값을 이렇게 설정한 이유 요약**   1. **eps=3**: 주어진 좌표 데이터가 특정 범위 안에서 비교적 조밀하게 분포되어 있으며, 반경 3 내에서 군집을 감지하는 것이 적합할 것으로 판단했습니다. 2. **min\_samples=5**: 군집이 형성되기 위해 요구되는 최소 밀도(5개 포인트)가 적절하다고 보았습니다. 이는 노이즈를 효과적으로 제외하면서 군집을 탐지하기 위함입니다.   **eps와 min\_samples 값 조정 가이드**   * **eps를 조정하는 방법**:   + eps를 너무 작게 설정하면 군집이 작고 조각조각 나뉩니다.   + eps를 너무 크게 설정하면 군집이 너무 커지고, 서로 다른 군집이 합쳐질 수 있습니다.   + **k-거리 그래프**를 사용하여 적절한 eps를 선택할 수 있습니다.     - 그래프에서 거리가 급격히 증가하는 지점을 선택하면 적절한 eps 값이 됩니다. * **min\_samples를 조정하는 방법**:   + min\_samples를 낮추면 군집이 더 쉽게 형성되지만, 노이즈 데이터가 군집으로 포함될 가능성이 높습니다.   + min\_samples를 높이면 데이터 밀도가 높은 군집만 형성되므로 노이즈 제거가 효과적입니다.   **데이터 특성에 따라 eps와 min\_samples를 재설정해야 하는 경우**   * 데이터가 더 조밀하게 분포했다면:   + eps를 줄이고, min\_samples도 줄입니다. * 데이터가 불규칙하고 넓게 분포했다면:   + eps를 늘리고, min\_samples를 높입니다. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1. eps 설정 (군집 반경)**  **eps란?**   * DBSCAN에서 eps는 두 점 간의 최대 거리로, 이 값 이내에 다른 점이 있으면 같은 군집으로 판단합니다. * 즉, **eps는 군집의 밀도 범위를 결정**합니다.   **eps를 설정하는 방법**   1. **직선의 점 간 거리 확인**:    * 필터링된 **파란 점들의 평균 거리**를 계산하여 적절한 eps 값을 추정합니다.    * 예를 들어, 점들의 평균 간격이 2라면, eps=2~3으로 설정해볼 수 있습니다. 2. **최적의 eps 찾기**:    * 데이터를 시각화하여 점들이 얼마나 밀집되어 있는지 확인합니다.    * 아래 코드를 사용하여 점 간 평균 거리와 분포를 확인할 수 있습니다:   python  코드 복사  from scipy.spatial.distance import pdist, squareform  # 필터링된 점들  filtered\_points = np.array(filtered\_points)  # 점 간 거리 계산  distances = pdist(filtered\_points) # 모든 점 간 거리  print("점 간 최소 거리:", distances.min())  print("점 간 평균 거리:", distances.mean())  print("점 간 최대 거리:", distances.max())   * 일반적으로:   + 점 간 **최소 거리 ~ 평균 거리** 사이의 값을 eps로 설정합니다.   + 예: 점 간 평균 거리가 2.5라면, eps=3 정도로 설정.  1. **추천 eps 값**:    * 상단 직선 데이터처럼 밀집된 경우: **eps=2~3**.    * 분포가 더 퍼져 있다면: **eps=3~5**.   **2. min\_samples 설정 (군집 최소 점 개수)**  **min\_samples란?**   * DBSCAN에서 한 군집으로 판단되기 위해 **최소 몇 개의 점**이 있어야 하는지 결정합니다. * min\_samples는 데이터의 밀도를 반영하며, 직선 패턴에서는 직선을 구성할 최소 점 개수를 기준으로 설정해야 합니다.   **min\_samples를 설정하는 방법**   1. **직선에 포함된 점 개수 추정**:    * 파란 점들이 직선에 몇 개 포함되어 있는지 확인합니다.    * 허프 변환으로 검출된 직선 위에 있는 점들이 적다면, min\_samples 값을 낮게 설정합니다. 2. **권장 값**:    * 상단 직선에서 점의 개수가 5~10개라면:      + \*\*min\_samples=3~5\*\*로 설정.    * 점의 개수가 10개 이상이라면:      + \*\*min\_samples=5~10\*\*으로 설정. 3. **노이즈를 줄이고 싶다면**:    * min\_samples를 높게 설정하여 군집화가 더 엄격하게 동작하도록 만듭니다.   **3. eps와 min\_samples의 조합 테스트**  DBSCAN에서 eps와 min\_samples의 조합을 다양하게 테스트하여 최적의 군집화를 찾는 것이 중요합니다.  **테스트 코드 예시**  python  코드 복사  # eps와 min\_samples 조합 테스트  for eps in [1, 2, 3, 4]:  for min\_samples in [2, 3, 5]:  cluster\_medians, labels = cluster\_and\_get\_medians(filtered\_points, eps=eps, min\_samples=min\_samples)  print(f"eps={eps}, min\_samples={min\_samples}: Cluster Medians: {cluster\_medians}")   * 이 코드는 여러 eps와 min\_samples 조합을 시도하여 결과를 비교합니다.   **4. 추천 설정 값**   1. **상단 직선 데이터처럼 밀집된 경우**:    * eps=2~3, min\_samples=3~5 2. **더 넓게 퍼진 데이터**:    * eps=4~5, min\_samples=5~7 3. **노이즈가 많은 경우**:    * eps를 줄이고(eps=2), min\_samples를 높게 설정(min\_samples=5~7).   **5. 데이터 기반 최적화**   1. **k-거리 그래프를 사용한 eps 최적화**:    * eps를 자동으로 최적화하기 위해 **k-distance graph**를 사용합니다.    * DBSCAN 라이브러리에서 k-distance graph를 그려 최적의 eps를 찾습니다.   **k-거리 그래프 코드**  python  코드 복사  from sklearn.neighbors import NearestNeighbors  import matplotlib.pyplot as plt  # k-거리 그래프  k = 4 # min\_samples에 해당  nbrs = NearestNeighbors(n\_neighbors=k).fit(filtered\_points)  distances, indices = nbrs.kneighbors(filtered\_points)  # 거리 정렬  distances = np.sort(distances[:, k-1], axis=0)  plt.plot(distances)  plt.title("k-distance Graph")  plt.xlabel("Points sorted by distance")  plt.ylabel("k-distance")  plt.grid()  plt.show()   * 그래프에서 **거리 급격히 증가하는 지점**의 값이 최적의 eps입니다.   **결론**   1. **추천 조합**:    * **eps=2~3**, **min\_samples=3~5** 2. **k-거리 그래프 사용**:    * 최적의 eps를 자동으로 찾고, 이를 기준으로 min\_samples를 설정. 3. **파라미터 튜닝**:    * 위 코드를 사용해 여러 조합을 시도한 뒤, 가장 적합한 결과를 선택.   **Q1. 직선 패턴 외에 더 복잡한 패턴을 군집화하려면 어떤 알고리즘이 적합할까?** **Q2. k-거리 그래프를 활용한 최적화 후에도 노이즈가 많다면 어떻게 개선할 수 있을까?** **Q3. eps와 min\_samples 값이 데이터 크기에 따라 어떻게 비례적으로 변경되는지 분석해볼까?**  4o  **나의 말:**  "점 간 최소 거리: 1.0 점 간 평균 거리: 9.52458567974974 점 간 최대 거리: 26.570660511172846" 이렇게 떴어  **ChatGPT의 말:**  ChatGPT  점 간 거리 통계가 아래와 같이 나왔습니다:   * **점 간 최소 거리**: 1.0 * **점 간 평균 거리**: 약 9.5 * **점 간 최대 거리**: 약 26.57   이 통계를 바탕으로 **eps**와 **min\_samples** 값을 설정하는 방식을 구체적으로 설명하겠습니다.  **1. eps 설정**  **eps를 설정하는 기준**   1. **최소 거리**: 1.0    * 최소 거리는 데이터가 매우 밀집된 경우를 나타냅니다.    * eps=1은 지나치게 엄격하여 대부분의 점이 노이즈로 처리될 가능성이 큽니다. 2. **평균 거리**: 약 9.5    * 평균 거리 근처로 설정하면 적당히 유연한 군집화를 수행할 수 있습니다.    * 따라서 eps를 약 8~10으로 설정하는 것이 적절해 보입니다. 3. **최대 거리**: 약 26.57    * 데이터의 최대 거리까지 포함하면 군집이 너무 넓어질 수 있으므로, 평균 거리와 최대 거리 사이 값을 선택하는 것도 방법입니다.    * 예: eps=15   **추천 eps 값**   * 좁은 범위 군집화: **eps=8~10** * 넓은 범위 군집화: **eps=12~15**   **2. min\_samples 설정**  **min\_samples를 설정하는 기준**   1. **점들의 밀도**:    * 점 간 최소 거리가 1.0이고 평균 거리가 약 9.5라면, 군집이 형성될 가능성이 높습니다.    * 데이터에서 적어도 3~5개의 점이 연결될 수 있도록 설정하는 것이 적절합니다. 2. **노이즈 제거**:    * min\_samples 값을 높게 설정하면, 노이즈를 제거할 가능성이 높아집니다.    * 하지만 너무 높게 설정하면 실제 군집의 일부가 제거될 수 있습니다.   **추천 min\_samples 값**   * 일반적인 설정: **min\_samples=3~5** * 노이즈 제거 강화: **min\_samples=6~8**   **3. 최적 조합**  다음 조합으로 테스트를 진행할 것을 권장합니다:   | **eps** | **min\_samples** | | --- | --- | | 8 | 3 | | 8 | 5 | | 10 | 3 | | 10 | 5 | | 12 | 5 | | 15 | 5 |   **4. 다음 단계**  **테스트 코드**  python  코드 복사  for eps in [8, 10, 12, 15]:  for min\_samples in [3, 5]:  cluster\_medians, labels = cluster\_and\_get\_medians(filtered\_points, eps=eps, min\_samples=min\_samples)  print(f"eps={eps}, min\_samples={min\_samples} -> Cluster Medians: {cluster\_medians}")  **결과 분석**   * 위 코드를 실행하여 각각의 조합에 대해 군집화 결과를 확인합니다. * 군집의 중앙값 좌표(cluster\_medians)와 노이즈로 처리된 점들을 확인하여 가장 적합한 조합을 선택하세요.   **5. 예상 결과 해석**   * **eps=8, min\_samples=3**:   + 더 좁은 범위의 군집을 형성.   + 노이즈로 처리된 점들이 더 많을 가능성 있음. * **eps=12, min\_samples=5**:   + 군집 범위가 넓어지며, 더 많은 점들이 군집으로 포함될 가능성이 높음. * **eps=15, min\_samples=5**:   + 더 느슨한 군집화를 수행하며, 일부 노이즈가 포함될 가능성도 있음.   **결론**   1. **eps**:    * 좁은 군집을 원한다면 eps=8.    * 넓은 범위를 포함하려면 eps=12~15. 2. **min\_samples**:    * 노이즈를 줄이고 싶다면 min\_samples=5. 3. **테스트 결과를 비교**:    * 각 조합의 결과를 확인한 뒤, 군집화 결과와 노이즈 수준을 비교하여 최적의 값을 선택. |

|  |
| --- |
| 선형회귀를 적용하여 filtered\_welding\_points를 구하려면, 먼저 welding\_points 데이터를 선형 회귀 모델에 적합한 형태로 처리하고, x 좌표를 예측값과 비교해 필터링하는 방식으로 접근해야 합니다.  아래는 선형회귀를 적용한 코드 수정 및 구현 예제입니다:  **1. 선형회귀 기반 필터링 로직**   * **목표**: 기존의 절대차 기준 필터링을 제거하고, 선형회귀 모델을 기반으로 이상치로 판단되는 점을 제거. * **방법**:   1. welding\_points 데이터를 사용해 y 좌표를 독립 변수로 하고, x 좌표를 종속 변수로 설정.   2. 선형회귀 모델을 학습시켜 예측값(predicted\_x) 생성.   3. 실제 x 값과 예측값의 차이가 특정 임계값을 초과하면 제거.   **2. 코드 구현**  python  코드 복사  import numpy as np  from sklearn.linear\_model import LinearRegression  import cv2  # welding\_points 리스트에 점들을 추가  welding\_points = []  for point in second\_points:  x\_coord, y\_coord = point  if overlap\_top\_peak <= y\_coord <= overlap\_btm\_peak:  welding\_points.append(point)  # 데이터 준비  welding\_points\_array = np.array(welding\_points)  x\_coords = welding\_points\_array[:, 0] # 종속 변수  y\_coords = welding\_points\_array[:, 1] # 독립 변수  # 2D 배열 형태로 변환  y\_coords\_reshaped = y\_coords.reshape(-1, 1)  # 선형 회귀 모델 학습  model = LinearRegression()  model.fit(y\_coords\_reshaped, x\_coords)  # x 좌표 예측  predicted\_x = model.predict(y\_coords\_reshaped)  # 임계값 기준으로 필터링 (예: ±5)  threshold = 5  filtered\_welding\_points = [  point for point, pred\_x in zip(welding\_points, predicted\_x)  if abs(point[0] - pred\_x) <= threshold  ]  # welding\_points를 y좌표 기준으로 정렬  welding\_points\_sorted = sorted(filtered\_welding\_points, key=lambda p: p[1])  # 점들을 연결하는 선 그리기  for i in range(len(welding\_points\_sorted) - 1):  pt1 = welding\_points\_sorted[i]  pt2 = welding\_points\_sorted[i + 1]  cv2.line(image\_with\_roi, pt1, pt2, (255, 0, 255), 2) # 보라색 선으로 연결  # 결과 시각화  cv2.imshow('ROI Visualization', image\_with\_roi)  cv2.waitKey(0)  cv2.destroyAllWindows()  **3. 코드 주요 변경점**   1. **필터링 로직 대체**: 기존의 절대차(abs(point[0] - second\_median[0]) <= 5) 기준 필터링 대신, 선형회귀 모델의 예측값과 실제 x 값의 차이를 활용. 2. **선형회귀 모델 학습**:    * y\_coords를 독립 변수로 설정 (y 값이 입력).    * x\_coords를 종속 변수로 설정 (x 값이 출력). 3. **임계값 설정**:    * 선형 모델의 예측값과 실제 값의 차이가 ±5를 초과하면 제거. 4. **선 정렬 및 그리기**:    * 필터링된 데이터를 y 좌표 기준으로 정렬한 후, 연결하여 시각화.   **4. 테스트 및 확장**   1. **임계값 조정**:    * threshold = 5는 경험적으로 설정된 값입니다. 데이터의 분포에 따라 조정 가능. 2. **시각화**:    * 선형 모델의 예측 결과를 함께 시각화하여 정확도를 확인할 수 있습니다. 3. **이상치 제거 시 확인**:    * filtered\_welding\_points 리스트에 포함되지 않은 좌표를 출력하여 어떤 데이터가 제거되었는지 확인 가능:   python  코드 복사  outliers = [point for point, pred\_x in zip(welding\_points, predicted\_x) if abs(point[0] - pred\_x) > threshold]  print("제거된 점들 (이상치):", outliers) |

|  |
| --- |
| 선형회귀에서 \*\*임계값(threshold)\*\*은 모델이 예측한 값과 실제 데이터 간의 \*\*오차(residual)\*\*를 기준으로 이상치(outlier)를 판별하거나 데이터를 필터링하는 데 사용됩니다. 임계값은 데이터 분포에 따라 조정하며, 일반적으로 다음과 같은 과정에서 사용됩니다:  **1. 오차(residual) 계산**  선형회귀 모델은 입력 데이터에 대해 예측값을 제공합니다. 예측값(predicted\_x)과 실제값(actual\_x)의 차이가 \*\*잔차(residual)\*\*입니다.  **공식**  residual=∣actual\_x−predicted\_x∣residual=∣actual\_x−predicted\_x∣  예:  python  코드 복사  residual = abs(actual\_x - predicted\_x)  **2. 임계값 정의**  잔차가 임계값을 초과하는 데이터는 이상치로 간주합니다. 임계값 설정 방법은 여러 가지가 있습니다.  **(1) 고정 임계값**   * 사용자가 경험적으로 설정하는 값입니다. * 예: 임계값을 threshold = 5로 설정하여 예측값과 실제값의 차이가 5를 넘으면 이상치로 간주.   **(2) 데이터 분포 기반 임계값**   * 데이터를 분석하여 동적으로 임계값을 결정합니다. * **평균(mean)** 및 \*\*표준편차(std)\*\*를 사용:threshold=mean(residual)+k×std(residual)threshold=mean(residual)+k×std(residual) 여기서 kk는 표준편차의 배수로, 일반적으로 k=2k=2나 k=3k=3을 사용.   python  코드 복사  # 임계값 계산 예제  threshold = 2 \* np.std(residuals) # 잔차의 표준편차의 2배를 임계값으로 설정  **3. 임계값으로 데이터 필터링**  잔차가 임계값을 초과하면 데이터를 제거하거나, 초과하지 않는 데이터를 필터링해 사용합니다.  **예제**  python  코드 복사  filtered\_points = [  point for point, pred\_x in zip(welding\_points, predicted\_x)  if abs(point[0] - pred\_x) <= threshold  ]  여기서 filtered\_points는 선형회귀 모델에 기반한 정상적인 데이터만 포함합니다.  **4. 임계값 활용의 실제 의미**  임계값을 통해 다음과 같은 작업을 수행할 수 있습니다:   1. **이상치 검출**: 데이터 중 선형 모델에서 크게 벗어난 점을 제거. 2. **노이즈 감소**: 측정 오차나 불필요한 변동성을 줄임. 3. **정확도 개선**: 선형 모델에 적합하지 않은 데이터 제거로 모델의 성능 향상.   **5. 임계값 적용 예제**  아래 코드는 welding\_points에서 임계값을 사용하여 이상치를 제거한 후 데이터를 필터링하는 과정을 보여줍니다.  python  코드 복사  # 잔차 계산  residuals = np.abs(x\_coords - predicted\_x)  # 임계값 설정 (예: 잔차의 표준편차를 기준으로)  threshold = 2 \* np.std(residuals)  # 임계값을 초과하지 않는 데이터 필터링  filtered\_welding\_points = [  point for point, pred\_x in zip(welding\_points, predicted\_x)  if abs(point[0] - pred\_x) <= threshold  ]  print("필터링된 점들:", filtered\_welding\_points)  **6. 임계값 설정 시 고려사항**   * **너무 낮은 값**: 정상 데이터가 제거될 수 있음. * **너무 높은 값**: 이상치가 포함될 수 있음. * 데이터의 특성을 파악하고 적절한 값을 선택하는 것이 중요합니다. 데이터의 분포를 시각화(예: 히스토그램, box plot)하여 적절한 임계값을 설정할 수 있습니다. |

|  |
| --- |
| 이 코드는 주어진 second\_points 데이터를 기반으로 특정 범위 내에서 데이터를 선별하고, 선형 회귀를 적용하여 이상치를 제거한 뒤, 필터링된 데이터를 시각화하는 과정입니다. 주요 단계와 각 코드의 역할을 자세히 살펴보겠습니다.  **1. overlap\_top\_peak와 overlap\_btm\_peak 조정**  python  코드 복사  overlap\_top\_peak += 20  overlap\_btm\_peak -= 20   * **역할**: overlap\_top\_peak와 overlap\_btm\_peak의 범위를 확장하여 y 좌표의 조건 범위를 넓힙니다.   + overlap\_top\_peak는 상한선을 위로 20만큼 증가.   + overlap\_btm\_peak는 하한선을 아래로 20만큼 감소. * **목적**: 데이터를 더 넓은 범위에서 검토할 수 있도록 함.   **2. welding\_points 리스트 생성**  python  코드 복사  welding\_points = []  for point in second\_points:  x\_coord, y\_coord = point  if overlap\_top\_peak <= y\_coord <= overlap\_btm\_peak:  welding\_points.append(point)   * **역할**: second\_points에서 y 좌표가 overlap\_top\_peak와 overlap\_btm\_peak 사이에 있는 점들만 선택하여 welding\_points 리스트에 추가. * **목적**: 관심 범위 내의 점들만 선별. * **결과**: welding\_points에는 (x, y) 좌표 형태의 데이터가 저장.   **3. 데이터 준비**  python  코드 복사  welding\_points\_array = np.array(welding\_points)  x\_coords = welding\_points\_array[:, 0] # 종속 변수  y\_coords = welding\_points\_array[:, 1] # 독립 변수   * **역할**:   + welding\_points를 NumPy 배열로 변환.   + x\_coords: x 좌표 값만 추출 (종속 변수, 선형회귀의 출력).   + y\_coords: y 좌표 값만 추출 (독립 변수, 선형회귀의 입력). * **목적**: 데이터를 선형 회귀 모델 학습에 적합한 형태로 변환.   **4. 2D 배열로 변환**  python  코드 복사  y\_coords\_reshaped = y\_coords.reshape(-1, 1)   * **역할**: y\_coords를 2D 배열로 변환 (모델 학습 요구사항). * **목적**: Scikit-learn의 LinearRegression 모델에 입력하기 적합한 형태로 변환.   **5. 선형 회귀 모델 학습**  python  코드 복사  model = LinearRegression()  model.fit(y\_coords\_reshaped, x\_coords)   * **역할**:   + y\_coords를 독립 변수로, x\_coords를 종속 변수로 설정.   + 선형회귀 모델을 학습하여 x와 y의 선형 관계를 학습. * **목적**: y 좌표로부터 x 좌표를 예측하는 모델 생성.   **6. x 좌표 예측**  python  코드 복사  predicted\_x = model.predict(y\_coords\_reshaped)   * **역할**: 학습된 선형 회귀 모델을 사용해 각 y\_coords에 대해 예측된 x 좌표(predicted\_x)를 계산. * **목적**: 모델이 예상하는 선형 관계에 기반한 x 좌표 생성.   **7. 이상치 제거**  python  코드 복사  threshold = 5  filtered\_welding\_points = [  point for point, pred\_x in zip(welding\_points, predicted\_x)  if abs(point[0] - pred\_x) <= threshold  ]  outliers = [point for point, pred\_x in zip(welding\_points, predicted\_x) if abs(point[0] - pred\_x) > threshold]  print("제거된 점들 (이상치):", outliers)   * **역할**:   + 예측된 x 값(predicted\_x)과 실제 x 값(point[0])의 차이를 계산.   + 차이가 threshold(±5) 이내인 점만 filtered\_welding\_points에 포함.   + 차이가 임계값을 초과한 점은 outliers 리스트에 저장. * **목적**: 선형 회귀 모델과 크게 벗어난 데이터를 제거하여 이상치를 정제.   **8. 정렬**  python  코드 복사  welding\_points\_sorted = sorted(filtered\_welding\_points, key=lambda p: p[1])   * **역할**: filtered\_welding\_points를 y 좌표를 기준으로 오름차순 정렬. * **목적**: 점들을 y 좌표 기준으로 순차적으로 연결할 수 있도록 정렬.   **9. 점 연결 및 시각화**  python  코드 복사  for i in range(len(welding\_points\_sorted) - 1):  pt1 = welding\_points\_sorted[i]  pt2 = welding\_points\_sorted[i + 1]  cv2.line(image\_with\_roi, pt1, pt2, (255, 0, 255), 2) # 보라색 선으로 연결  cv2.imshow('ROI Visualization', image\_with\_roi)  cv2.waitKey(0)  cv2.destroyAllWindows()   * **역할**:   + 정렬된 welding\_points\_sorted 리스트에서 인접한 점들 사이를 직선으로 연결.   + OpenCV를 사용하여 이미지(image\_with\_roi) 위에 보라색 선을 그립니다. * **목적**: 필터링 및 정렬된 점들을 시각적으로 확인.   **최종 흐름 요약**   1. 관심 범위(overlap\_top\_peak, overlap\_btm\_peak) 내 데이터를 선별하여 welding\_points 리스트 생성. 2. 선형회귀 모델을 학습하여 y로부터 x를 예측. 3. 예측값과 실제값의 차이가 임계값(±5)을 초과하는 점들을 제거. 4. 필터링된 데이터를 y 기준으로 정렬. 5. 정렬된 점들을 직선으로 연결하여 시각화. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 좋은 질문입니다. 기존에 중앙 좌표(second\_median)를 기준으로 x좌표가 ±5 이상 차이나면 제외한 방식과 선형회귀 기반 필터링 방식의 차이는 **적응성**, **데이터 분포를 고려한 유연성**, **정확성**에서 드러납니다.  **차이점 비교**  **1. 기존 방식 (중앙 좌표 ±5 기준 필터링)**   * **작동 원리**:   + 모든 데이터에서 second\_median[0]을 기준으로 x좌표의 절대 차이가 ±5를 넘는 데이터를 제거. * **장점**:   + 계산이 단순하며 빠름.   + 중앙값만으로 간단히 데이터 필터링 가능. * **단점**:   + 데이터 분포를 고려하지 않음. 예를 들어, x좌표가 선형적인 경향을 가지더라도 중앙값 기준으로 ±5를 초과하면 제거됨.   + 모든 데이터에 동일한 기준을 적용하므로, 데이터가 선형적 또는 비선형적 경향을 가질 때 정확히 반영하지 못함.   **2. 선형회귀 기반 필터링**   * **작동 원리**:   + y좌표와 x좌표의 관계를 학습하여 x좌표의 예측값(predicted\_x)을 계산.   + predicted\_x와 실제 x좌표의 차이를 기준으로 필터링. * **장점**:   + 데이터 분포를 학습하고 반영하여 동적으로 x좌표를 평가.   + x좌표와 y좌표의 상관관계가 있으면 더 정확한 필터링 가능.   + 데이터가 선형적 경향을 가질 경우 더 높은 정확성을 보장. * **단점**:   + 선형회귀 학습 및 계산이 추가되므로 약간 더 복잡.   + 데이터가 완전히 선형적이지 않거나 패턴이 복잡할 경우 비선형 회귀가 필요할 수 있음.   **주요 차이점**   | **기준** | **중앙 좌표 ±5 방식** | **선형회귀 방식** | | --- | --- | --- | | **기준** | second\_median[0] ± 5 | predicted\_x ± threshold | | **데이터 반영** | 고정된 중앙값 기준 | 데이터의 선형 관계를 학습하여 동적 기준 생성 | | **적용 상황** | 데이터 분포와 관계없이 단순히 절대적 거리 비교 | 데이터 분포와 선형 경향성을 반영한 필터링 | | **유연성** | 낮음: 데이터의 경향을 반영하지 못함 | 높음: 데이터의 경향에 따라 임계값이 조정 가능 | | **필터링 정확성** | 데이터 분포가 비선형적일 때도 제거 가능 | 데이터가 선형적일 경우 더 정확하게 필터링 가능 | | **계산 복잡도** | 간단 | 상대적으로 복잡 |   **구체적 예**  **중앙 좌표 기반 방식**   * 예를 들어, 데이터 분포가 아래와 같을 경우:   text  코드 복사  x좌표: [100, 105, 120, 200, 300]  중앙값(second\_median[0]): 120   * + x좌표 200과 300은 중앙값 ±5 기준에서 벗어나 제거됨.   + 하지만 x좌표 105와 120 사이의 관계나 경향은 고려되지 않음.   **선형회귀 방식**   * 동일한 데이터 분포에서, y좌표와 x좌표의 선형 관계를 학습하여 predicted\_x를 계산:   text  코드 복사  predicted\_x: [100, 110, 130, 210, 320]   * + x좌표 200은 예측값 210과의 차이가 임계값 ±5를 초과하지 않아 유지될 수 있음.   + 데이터의 경향(선형적 패턴)을 고려하므로, 이상치 제거가 더 정확.   **언제 선형회귀가 더 유리한가?**   1. **x좌표와 y좌표 간에 상관관계가 있을 때**:    * 데이터가 선형적 경향을 가질 경우, 선형회귀를 적용하면 중앙값 기준보다 더 정밀한 필터링 가능. 2. **데이터가 다양하고 동적으로 변화할 때**:    * 중앙값이 변하기 어렵거나 데이터 분포가 일정하지 않은 경우, 선형회귀는 데이터 경향성을 반영해 동적 필터링 가능.   **결론**   * 중앙 좌표 기반 방식은 **간단하고 빠르지만**, 데이터가 선형적 경향을 보일 때 부정확한 필터링 가능성이 있음. * 선형회귀 기반 필터링은 **데이터의 분포와 경향을 반영**하므로 더 유연하고 정확하게 필터링할 수 있음. * 최적의 선택은 데이터의 특성에 따라 달라지며, 선형적 관계가 확실하다면 선형회귀가 더 나은 선택입니다. |