

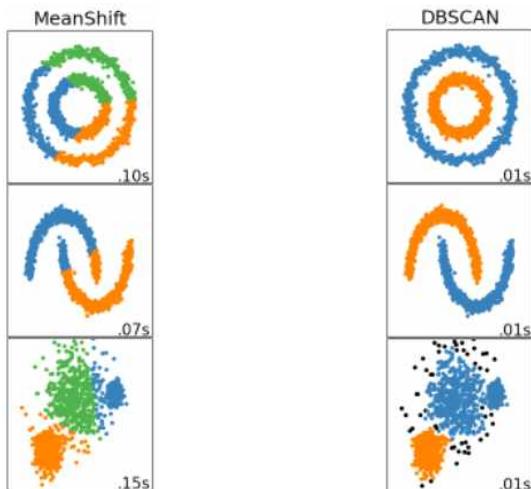
Big Data Project

2017147558 컴퓨터과학과 김경민

<Phase 1 : Geographical clustering>

1. 개요

Meanshift를 사용해서 clustering하였다. K-Means을 이용할 경우 cluster의 개수를 사전에 정해야하기 때문에 사용하지 않았다. Landmark의 특성상 Dense하게 뭉쳐있을 것이므로 Density based clustering을 사용하였다. DBSCAN의 경우 sample해서 사진을 뽑아보았을 때 Meanshift와 큰 차이가 없었지만, toy example을 살펴보면 장소에 대한 Density 자체보다 density한 data set들끼리 묶이는 경향이 있으므로 Meanshift를 선택하였다.



Bandwidth로 estimate_bandwidth 함수를 사용했고, quantile 값을 변경해가면서 clustering하고 clustering visualization과 cluster마다 랜덤한 sample 사진을 뽑아 결과를 확인했다. 이를 통해 적절한 quantile을 결정했다.

2. 이론

Window 내부의 data에 대해서 Kernel 함수를 이용해 weighted mean of density를 구하고 이러한 density가 높은 point로 window의 중심을 옮긴다. 이것을 converge할 때까지 반복한다. weighted mean of density를 구하는 공식은 다음과 같다.

$$m(x) = \frac{\sum_{x_i \in N(x)} K(x_i - x)x_i}{\sum_{x_i \in N(x)} K(x_i - x)}$$

3. 구현

```
quantile = 0.1
bandwidth = estimate_bandwidth(coordinate, quantile=quantile, n_samples=1000, n_jobs=-1)
cluster = MeanShift(bandwidth = bandwidth, bin_seeding=True, n_jobs = -1).fit(coordinate)
```

sklearn의 MeanShift함수를 사용했다. bandwidth로는 estimate_bandwidth를 사용했다. quantile 값을 변경해가면서 적절한 clustering을 찾았다. estimate_bandwidth에 n_sample 값을 주지 않으면 실행시간이 $O(N^2)$ 이 된다. 이번 task의 경우 N의 값이 100만에 가까워서 실행시간이 매우 길었다. 따라서 n_sample을 1000으로 설정했다. n_jobs을 -1으로 설정해서 프로세서를 최대로 활용했다. MeanShift 함수에서 bin_seeding을 True로 설정하지 않을 경우 initial kernal이 모든 point에 대해 결정하게 된다. 이 경우 실행시간이 12시간이 넘어도 clustering이 끝나지 않았기 때문에 bin_seeding을 True로 설정했다. 마찬가지로 프로세스를 최대로 활용하기 위해 n_jobs를 -1로 설정했다.

```
data = {'photo_id' : photos['photo_id'], 'cluster' : labels}
dataframe = pd.DataFrame(data)

cluster_list = {}
for i in range(num_clusters):
    cluster_list[i] = np.array(dataframe.loc[dataframe['cluster'] == i]['photo_id'].astype(str))

photo_id_list = []
path_photos = './Photos'
photofile_list = os.listdir(path_photos)
for photofile in photofile_list:
    photo_id_list.append(photofile[:-4])

photo_cluster = {}
for photo_id in photo_id_list:
    for key, value in cluster_list.items():
        if photo_id in value:
            if key in photo_cluster:
                photo_cluster[key].append(photo_id)
            else:
                photo_cluster[key] = [photo_id]
```

이후 cluster된 결과를 pandas dataframe을 이용해 합친 다음 Photos에 있는 사진 리스트와 비교하여 photo_cluster라는 딕셔너리에 cluster 별로 photo id를 저장했다.

```
path_photos = './Photos'
photofile_list = os.listdir(path_photos)

photo_id_list = []
for filename in photofile_list:
    photo_id = filename[:-4]
    photo_id_list.append(photo_id)

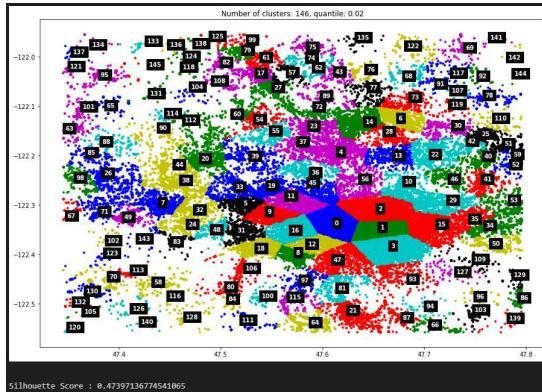
photo_cluster = {}
for i in range(len(labels)):
    id = photos['photo_id'][i]
    cluster = labels[i]
    if id in photo_id_list:
        img = cv.imread(path_photos + '/' + id + '.jpg')
        photo_cluster[cluster].append(img)
```

원래는 위와 같은 구조를 사용했으나, 100만 loop에서 4천개의 find를 하는 것이 실행시간이 오래 걸리는 것을 발견했다. 먼저 cluster 별로 나눈 다음 각 cluster에 실제있는 사진을 find하는 방식으로 실행시간을 단축시켰다.

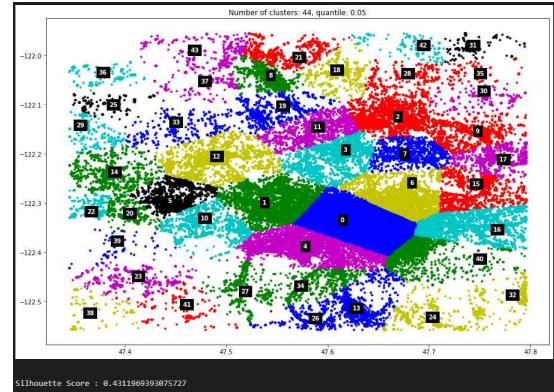
4. 결과 및 분석

matplotlib의 pyplot을 이용해 cluster를 가시화했다. Silhouette Score도 구해서 출력했다. quatine을 변경해가면서 clustering이 어떻게 진행되는지 실험했다. 결과는 다음과 같다.

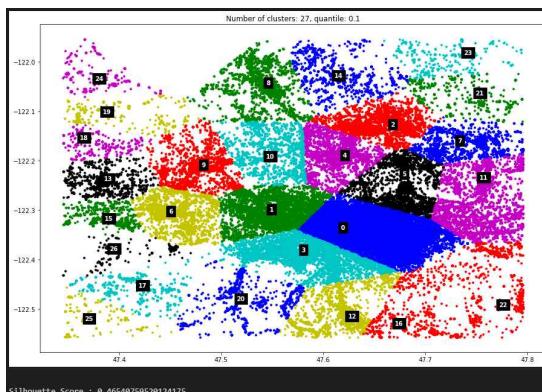
1) quatine : 0.02



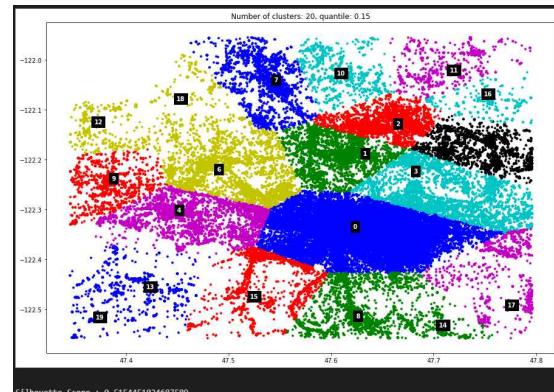
2) quatine : 0.05



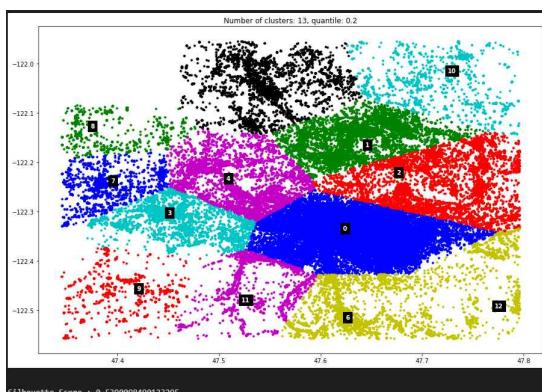
3) quatine : 0.1



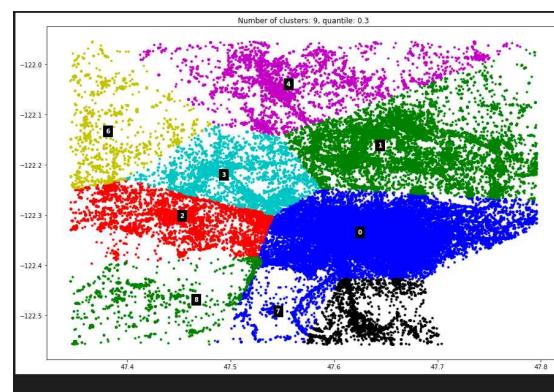
4) quatine : 0.15

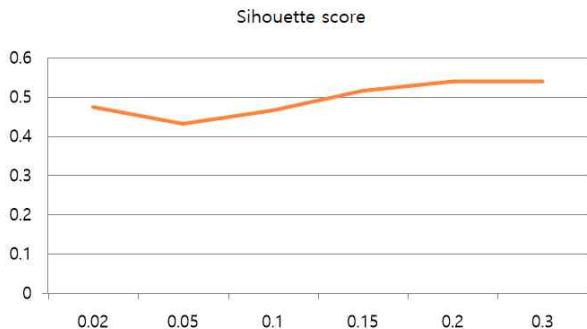


5) quatine : 0.2



6) quatine : 0.3

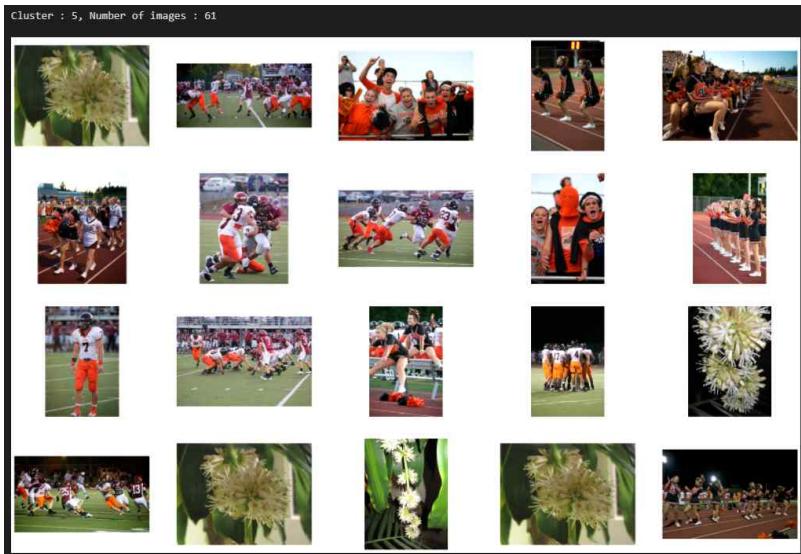




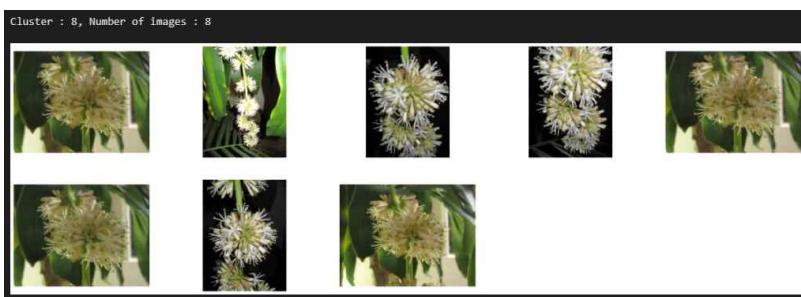
Clustering의 지표로서 Silhouette score을 사용하려 했으나, 잘 clustering된 것으로 보이는 것에 더 낮은 score가 할당됐다. 이에 따라 Silhouette score을 지표로 사용하지 않았다.

이후 clustering된 실제 사진들을 random sample하여 20개씩 나열해 clustering 경과를 관찰했다.

quantile : 0.2



quantile : 0.1



quantile이 0.2인 경우에는 quantile이 0.1일 때 잘 분류되던 사진들이 같이 clustering되었다.

quantile : 0.05

Cluster : 0, Number of images : 2993

quantile : 0.1

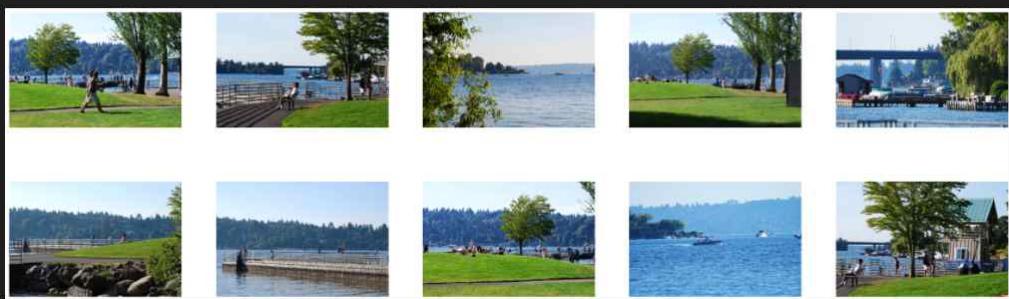
Cluster : 0, Number of images : 3341

quantile이 0.05일 때는 더 세분화하게 clustering 되었지만 정작 0th cluster의 사진 개수가 2993개로 3341개인 quantile : 0.1과 크게 차이 나지 않았다. 더 작게 clustering 될수록 같은 landmark가 다른 cluster에 포함될 가능성이 높으므로 quantile 값은 0.1로 결정하였다. quantile 값은 0.05 ~ 0.15가 적정 범위로 보인다.

quantile 값을 조정해 geo cluster를 세분화하는 것은 특정 지역에 point가 밀집된 이번 task에 맞지 않았다. quantile을 작게 해서 cluster의 수가 늘어나면 point가 많은 cluster가 세분화하게 나뉘는 것이 아니라 작은 cluster가 더 작은 cluster로 나뉘는 것을 확인했다. 대다수의 경우에서 quantile 값 자체를 작게 해서 잘게 나누는 것보다 clustering된 것을 다시 clustering하는 것이 잘 분류되었다. 다음 단계에서 이를 적용할 예정이다.

다음은 quantile : 0.1일 때 잘 clustering된 예시이다.

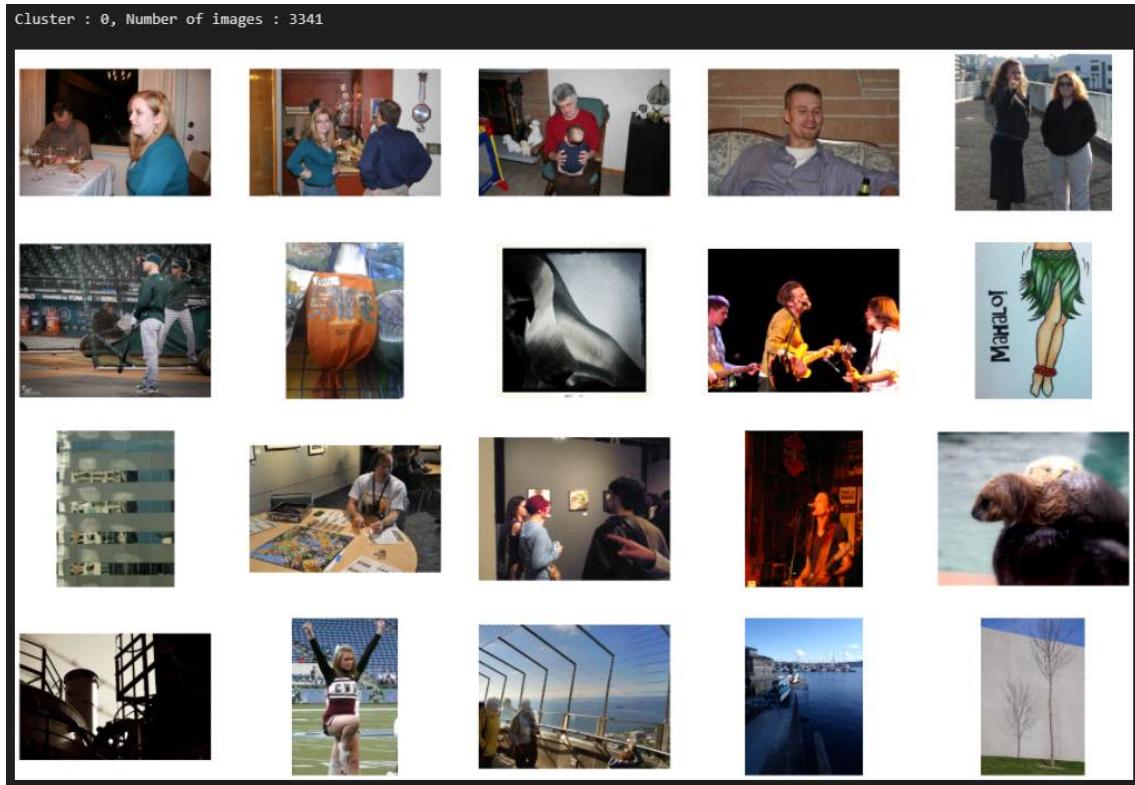
Cluster : 10, Number of images : 18



Cluster : 14, Number of images : 53



반면 사진 개수가 많은 cluster : 0은 사진에 일관성이 보이지 않았고, 너무 밀집되어 있었다.



<Phase 1-2 : Geographical Clustering recursively>

1. 개요

Geographical clustering을 하는 과정에서 특정 지역에 points들이 밀집되어 있는 것을 확인했다. 이러한 경우에 한 번에 작은 cluster로 나누는 것보다 clustering을 여러 번 하는 것이 더 효과적이라는 것을 실험적으로 발견했다.

원래는 Geo-cluster에 의해 적당히 나눠진 cluster에 대해 visual clustering을 진행하여 landmark를 구분하려 했으나, 100개 정도의 data 셋에 visual clustering을 적용했을 때도 실행시간이 12시간 정도 걸렸다. 모든 data에 대해 pairwise한 similarity를 계산하는 것은 $\frac{n(n+1)}{2}$ 의 실행을 요구하므로 cluster의 크기를 줄일 필요가 있었다. 이에 반복적으로 Geo clustering을 적용해서 밀집되어 있는 point들을 분류했다. 처음에는 한 step만 더 clustering 하려 했으나, 결과가 첫 번째와 마찬가지로 한 클러스터에 치우친 결과가 나와서 recursive하게 적용했다.

모든 cluster의 사이즈가 100이하가 될 때까지 recursive하게 clustering을 적용했다. 이후 결과가 확연하게 좋지 않은 경우에는 quantile 값을 변경해가면서 수정했다.

2. 구현

```
def more_cluster(cluster_id_list, quantile):
    filename = "_".join([str(i) for i in cluster_id_list])
    dataset = pd.read_csv(f"csv_result/{filename}.csv", header=0, index_col=0)

def move_cluster(cluster_id, photo_cluster):
    for key, value in photo_cluster.items():
        for id in value:
            copy_dir = f'./geo_cluster/{cluster_id}_{key}'
            if not os.path.isdir(copy_dir):
                os.makedirs(copy_dir)
            shutil.copy(f'./geo_cluster/{cluster_id}/{id}.jpg', f'{copy_dir}/{id}.jpg')

for i in range(num_clusters):
    tmp_dataframe_csv = dataframe_csv[dataframe_csv['cluster'] == i]
    tmp_dataframe_csv.to_csv(f"csv_result/0_0_{i}.csv", mode='w')

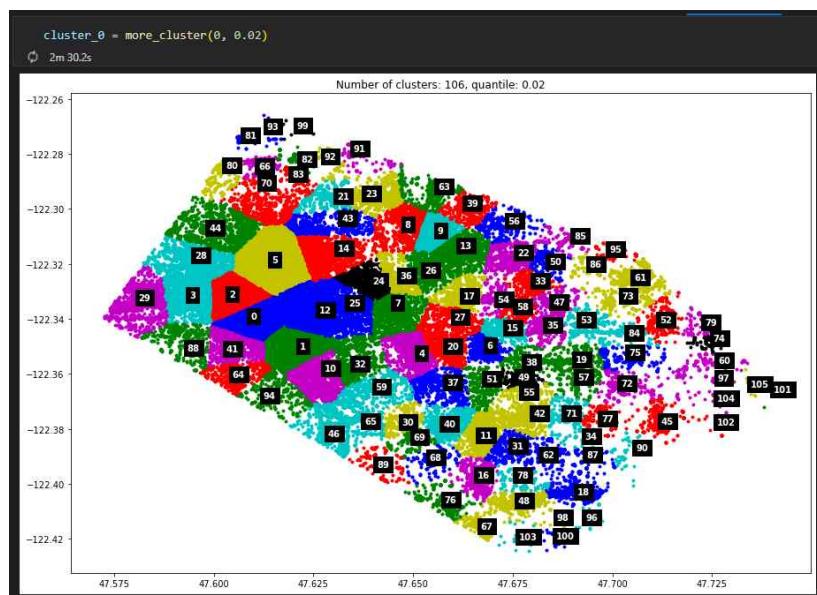
move_cluster('0_0_0', cluster)
```

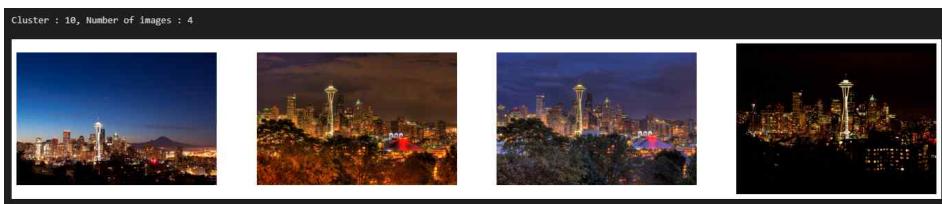
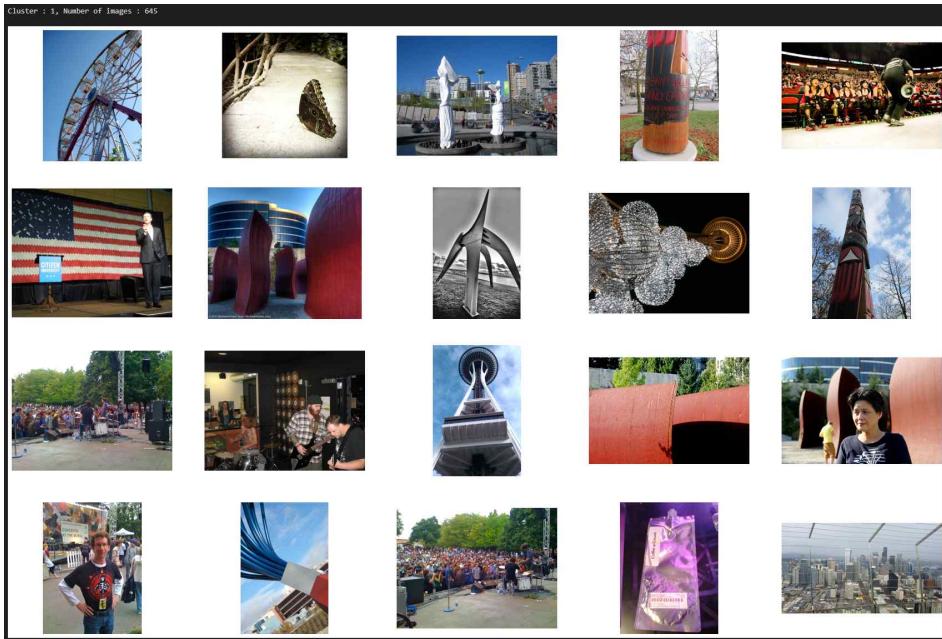
처음에는 위에서 구한 dataframe을 사용해서 구현했으나, depth가 깊어짐에 따라 매번 이전 dataframe을 구하는 것이 overhead가 커졌다. 이에 csv파일을 읽고 쓰는 방식으로 변경했다. csv 파일 형식은 {cluster_id1}_{cluster_id2}_...{cluster_idn}이다. 각 cluster_id는 depth별 clustering 결과 값이다. 이후 cluster에 따라 사진을 다른 폴더로 copy했다. 이 경우에도 사진 폴더의 형식은 csv와 같다.

함수도 list 형식을 input으로 받아 범용적으로 작동하도록 수정했다. 나머지 함수는 위에서 사용한 코드를 이용하였다.

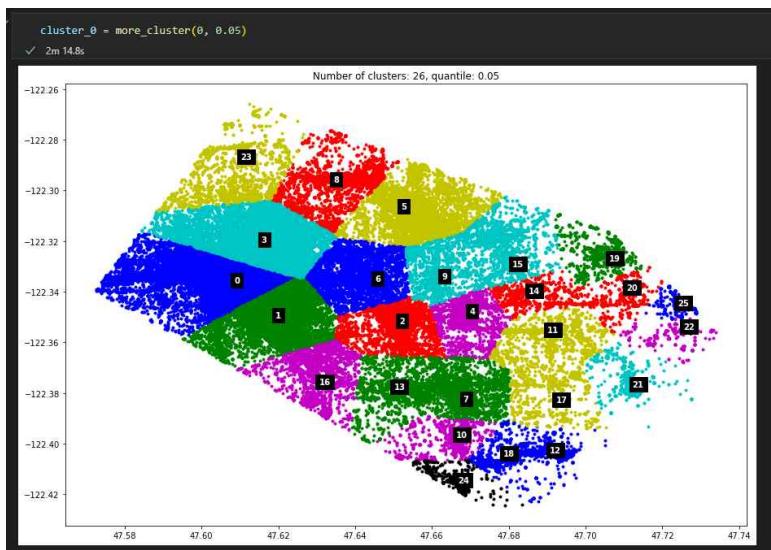
3. 결과 및 분석

적절한 quantile 값을 찾기 위해 실험을 반복했다.

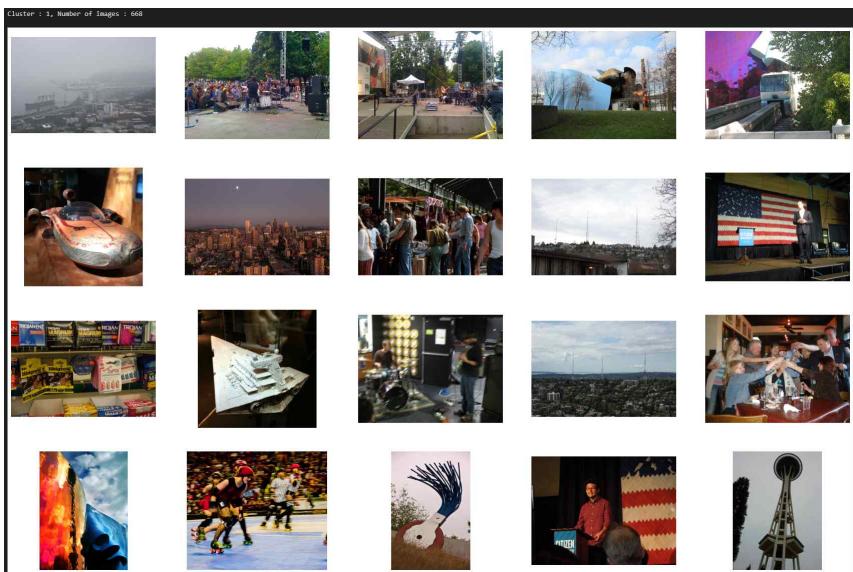
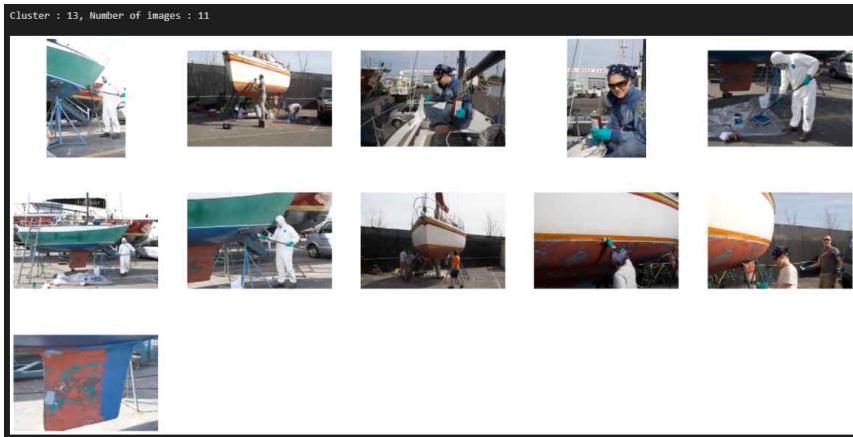




이번에도 마찬가지로 quantile 값을 줄여 작게 clustering 하는 것은 도움이 되지 않았다. 작게 clustering 했더니 landmark인 타워가 다른 cluster에 포함됐음을 확인할 수 있다.

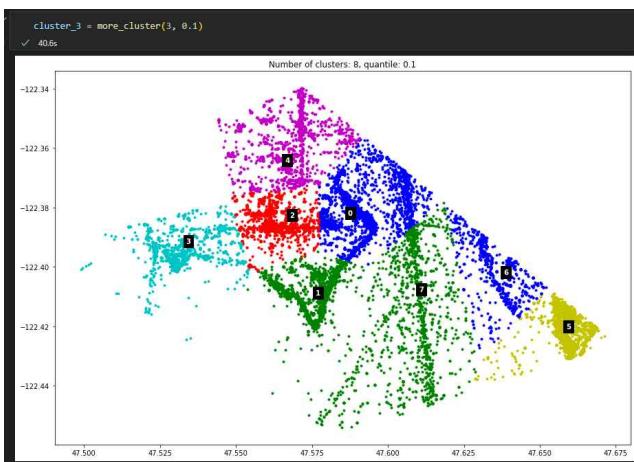


실험 결과 cluster 0에 대해 적절한 quantile 값은 0.05로 판단된다. 다음은 clustering이 잘 된 예시이다.



하지만 첫 번째 step과 마찬가지로 특정 cluster에 많은 사진이 쏠렸다. 이를 해결하기 위해 clustering을 recursive하게 여러 번 했다. stop point는 모든 cluster의 크기가 100이 하일 때이다.

다음은 recursive하게 clustering한 결과 중 cluster 3를 다시 clustering한 결과이다.



Cluster : 6, Number of images : 1



Cluster : 7, Number of images : 2



Cluster : 0, Number of images : 2

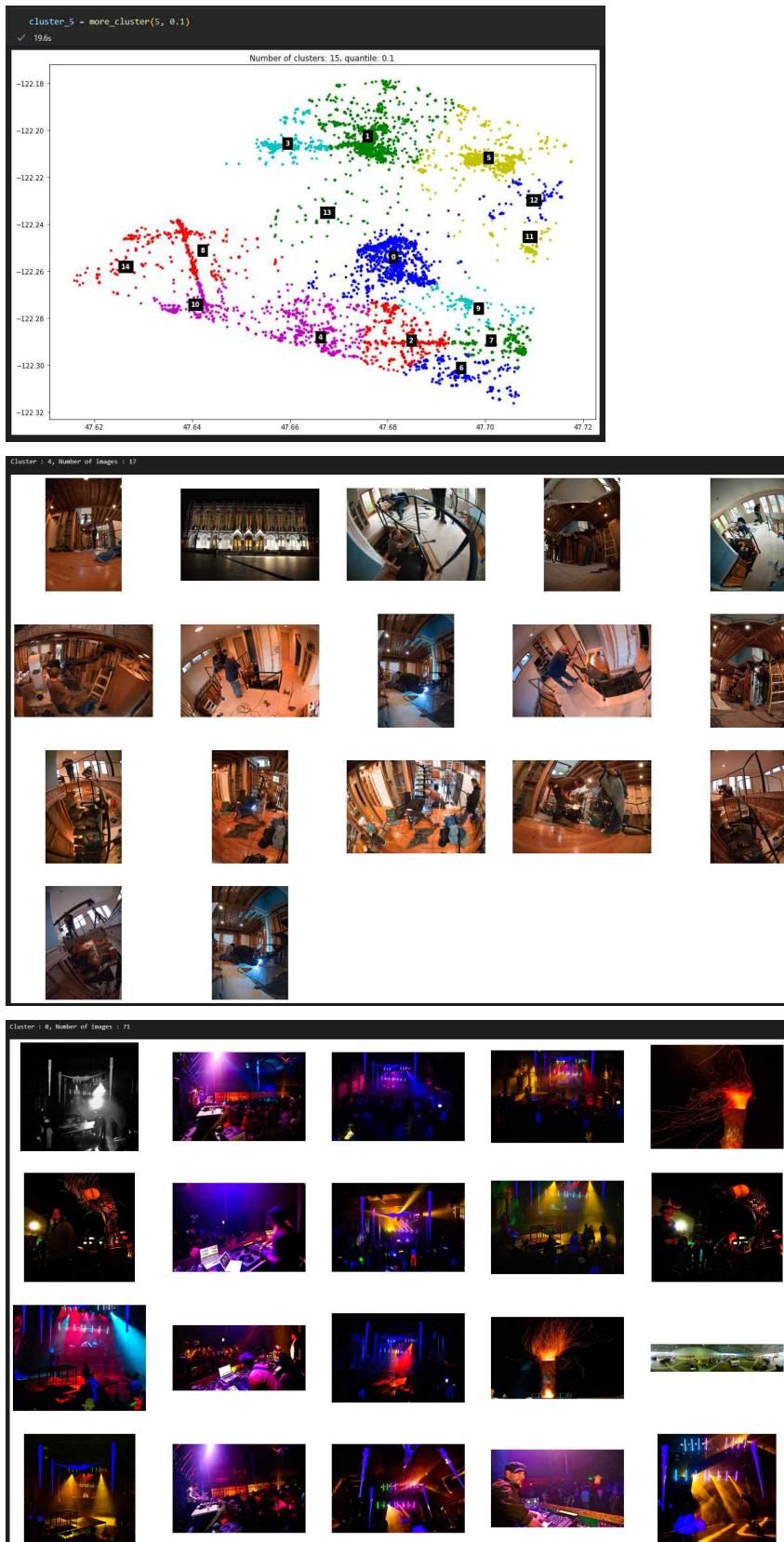


recursive하게 cluster를 나누다 보니 위와 같이 같은 landmark에 대해 다른 cluster로 분류되었다.

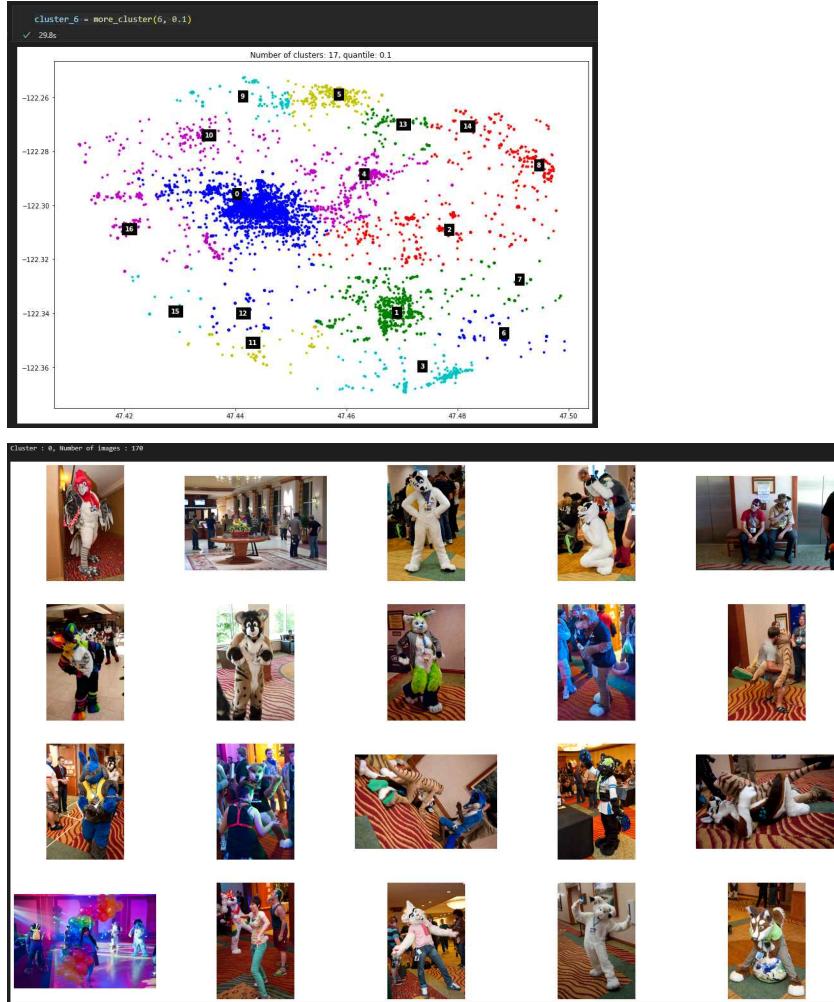
더 큰 quantile을 써서 geo clustering을 진행했지만 quantile이 조금만 커져도 세부 clustering이 되지 않았다. quantile 값을 조정해서는 clustering이 잘 되지 않았다. cluster의 수가 190으로 컸지만 전부 visual clustering 하였다.

다음은 recursive하게 clustering이 잘 된 예시이다.

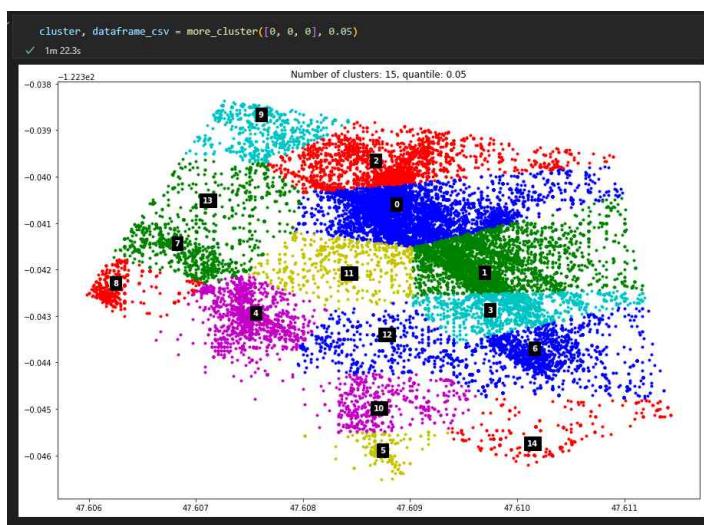
case 1)

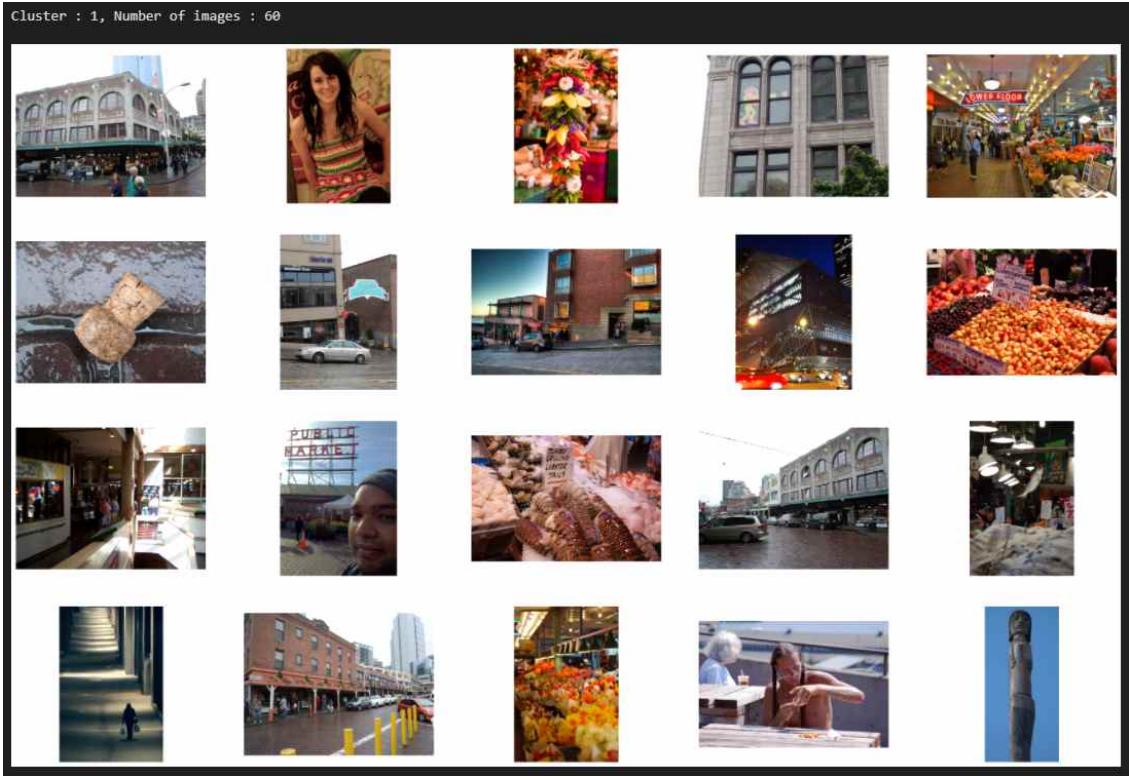


case 2)



recursive하게 하다보니 cluster안 사진 수 60으로 작은데도 여러 landmark가 섞여 있는 cluster도 발견했다.





이러한 경우에 대해 Visual clustering을 적용해 landmark 끼리 나눌 예정이다.

상당히 많은 cluster들이 적절한 사진의 셋으로 잘 나뉘어진 것을 확인할 수 있었다. 아래의 표는 recursive하게 geo clustering을 마친 결과이다. 총 96개의 cluster가 완성되었다.

1st	2nd	3rd	4th
			0
			1
			2
			3
		0	4
			6
			7
			8
			9
			11
			13
		1	0
			1
			2
			4
0		2	0
0			1
0			2
0			3
0			4
0			6
0			9
0		3	0
0			4
0		5	0
0			1
0			2
0			3
0			4
0			5
0		6	0
0			1
0			2
0			3
0			4
0			5
0		7	0
0			1
0			2
0			3
0			4
0		8	0
0			1
0			2
0			3
0			4
0		9	0
0			1
0			2
0			3
0			4
0		10	0
0			1
0			2
0			3
0		11	0
0			1
0			2
0			3
0		17	0
0			1
0			2
0		23	0
0			1
0			2
0			3
0			4
0		21	0
0			1
0			2
0			3
0			4
0			5
0		22	0
0			1
0			2
0			3
0			4
0			5
0		23	0
0			1
0			2
0			3
0			4
0		24	0
0			1
0			2
0			3
0			4
0		25	0
0			1
0			2
0			3
0			4
0		26	0
0			1
0			2
0			3
0			4
0		27	0
0			1
0			2
0			3
0			4
0		28	0
0			1
0			2
0			3
0			4
0		29	0
0			1
0			2
0			3
0			4
0		30	0
0			1
0			2
0			3
0			4
0		31	0
0			1
0			2
0			3
0			4
0		32	0
0			1
0			2
0			3
0			4
0		33	0
0			1
0			2
0			3
0			4
0		34	0
0			1
0			2
0			3
0			4
0		35	0
0			1
0			2
0			3
0			4
0		36	0
0			1
0			2
0			3
0			4
0		37	0
0			1
0			2
0			3
0			4
0		38	0
0			1
0			2
0			3
0			4
0		39	0
0			1
0			2
0			3
0			4
0		40	0
0			1
0			2
0			3
0			4
0		41	0
0			1
0			2
0			3
0			4
0		42	0
0			1
0			2
0			3
0			4
0		43	0
0			1
0			2
0			3
0			4
0		44	0
0			1
0			2
0			3
0			4
0		45	0
0			1
0			2
0			3
0			4
0		46	0
0			1
0			2
0			3
0			4
0		47	0
0			1
0			2
0			3
0			4
0		48	0
0			1
0			2
0			3
0			4
0		49	0
0			1
0			2
0			3
0			4
0		50	0
0			1
0			2
0			3
0			4
0		51	0
0			1
0			2
0			3
0			4
0		52	0
0			1
0			2
0			3
0			4
0		53	0
0			1
0			2
0			3
0			4
0		54	0
0			1
0			2
0			3
0			4
0		55	0
0			1
0			2
0			3
0			4
0		56	0
0			1
0			2
0			3
0			4
0		57	0
0			1
0			2
0			3
0			4
0		58	0
0			1
0			2
0			3
0			4
0		59	0
0			1
0			2
0			3
0			4
0		60	0
0			1
0			2
0			3
0			4
0		61	0
0			1
0			2
0			3
0			4
0		62	0
0			1
0			2
0			3
0			4
0		63	0
0			1
0			2
0			3
0			4
0		64	0
0			1
0			2
0			3
0			4
0		65	0
0			1
0			2
0			3
0			4
0		66	0
0			1
0			2
0			3
0			4
0		67	0
0			1
0			2
0			3
0			4
0		68	0
0			1
0			2
0			3
0			4
0		69	0
0			1
0			2
0			3
0			4
0		70	0
0			1
0			2
0			3
0			4
0		71	0
0			1
0			2
0			3
0			4
0		72	0
0			1
0			2
0			3
0			4
0		73	0
0			1
0			2
0			3
0			4
0		74	0
0			1
0			2
0			3
0			4
0		75	0
0			1
0			2
0			3
0			4
0		76	0
0			1
0			2
0			3
0			4
0		77	0
0			1
0			2
0			3
0			4
0		78	0
0			1
0			2
0			3
0			4
0		79	0
0			1
0			2
0			3
0			4
0		80	0
0			1
0			2
0			3
0			4
0		81	0
0			1
0			2
0			3
0			4
0		82	0
0			1
0			2
0			3
0			4
0		83	0
0			1
0			2
0			3
0			4
0		84	0
0			1
0			2
0			3
0			4
0		85	0
0			1
0			2
0			3
0			4
0		86	0
0			1
0			2
0			3
0			4
0		87	0
0			1
0			2
0			3
0			4
0		88	0
0			1
0			2
0			3
0			4
0		89	0
0			1
0			2
0			3
0			4
0		90	0
0			1
0			2
0			3
0			4
0		91	0
0			1
0			2
0			3
0			4
0		92	0
0			1
0			2
0			3
0			4
0		93	0
0			1
0			2
0			3
0			4
0		94	0
0			1
0			2
0			3
0			4
0		95	0
0			1
0			2
0			3
0			4
0		96	

<Phase 2 : visual clustering>

1. 개요

Visual clustering을 위해 예시로 주어진 bundler를 이용하려 했으나, 다양한 환경에서 시도했음에도 성공하지 못하였다. 이에 Feature matching algorithm을 이용해 Visual clustering을 시도했다. 처음에는 SIFT와 SURF 중에서 SURF를 사용하려 했었다. SIFT에 비해 실행시간이 짧아 시간적 한계 때문에 cluster의 크기를 제한하는 현재 상황에 잘 맞는다고 판단했다. 하지만 SURF는 NONFREE Algorithm이었기 때문에 사용할 수 없었다. SIFT를 사용해서 분석했다.

SIFT가 다른 시간대의 사진을 잘 매칭하지 못한다는 점을 해결하기 위해 촬영 시간에 따라 시간 차이가 많이 나는 경우에만 grayscaling을 적용하려 했다. 그렇게 하면 grayscaling을 해서 얻은 SIFT matches와 grayscaling을 하지 않고 얻은 SIFT matches의 정량적인 비교가 불가능해서 전부 grayscaling을 하였다.

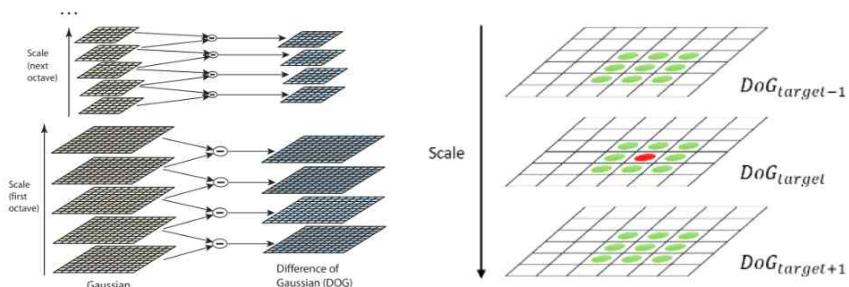
SIFT를 이용해서 pairwise한 similarity matrix를 계산했다. 이것을 바탕으로 connected graph에서 threshold를 적용한 cut을 통해 연결된 graph끼리 cluster를 구성했다.

2. 이론

SIFT란 Scale Invariant Feature Transform의 줄임말로써 이미지의 scale과 rotation에 robust한 feature을 추출하는 알고리즘이다. Scale-space의 생성을 위해 원본 이미지를 resizing해서 image pyramid를 만든다. 이후 각 층에 gaussian blurring을 적용한 이미지를 얻어 scale space를 만든다. 옆의 사진이 scale space의 예시이다.



같은 octave내의 인접한 gaussian blurred image 2개를 빼기 연산을 하여 DoG를 구한다. 각 fixel에 대해 인접한 3x3x3 cube와 비교해 extrema인지 판단한다. extrema 중 stable한 keypoint를 선정한다. 이렇게 되면 이러한 keypoint들은 scale에 invariance하게 된다. 이 과정은 아래 사진과 같다.



gaussian blurred image에서 key point를 중심으로 16x16 patch를 추출한다. 아래 식을 이용해 Gradient magnitude와 Gradient orientation을 계산한다.

$$m(x, y) = \sqrt{(L(x+1, y) - L(x-1, y))^2 + (L(x, y+1) - L(x, y-1))^2}$$

$$\theta(x, y) = \tan^{-1}((L(x, y+1) - L(x, y-1)) / (L(x+1, y) - L(x-1, y)))$$

4x4 sub-patch로 나눈 뒤, 이 sub-patch에 orientation histogram을 통해 8방향 정보를 저장한다. 이렇게 구한 descriptor에서 keypoint orientation을 뺀다. 이를 통해 rotation invariance 성질을 얻는다. 이러한 과정을 통해 SIFT를 통한 feature matching은 scale과 rotation에 대해 invariance한 성질을 갖는다.

3. 구현

```
def sift(cluster_id):
    dir = f'geo_cluster/{cluster_id}'
    photofile_list = os.listdir(dir)
    sift_similarity = np.zeros((len(photofile_list), len(photofile_list)))

    for i in range(len(photofile_list)-1):
        for j in range(i+1, len(photofile_list)):
            img1 = cv2.imread(dir + '/' + photofile_list[i])
            img2 = cv2.imread(dir + '/' + photofile_list[j])
            gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
            gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

            # SIFT detector
            detector = cv2.xfeatures2d.SIFT_create()
            kp1, desc1 = detector.detectAndCompute(gray1, None)
            kp2, desc2 = detector.detectAndCompute(gray2, None)

            # BFMatcher
            matcher = cv2.BFMatcher()
            # calculate matching
            matches = matcher.knnMatch(desc1, desc2, k=2)

            good = 0
            for m, n in matches:
                if m.distance < 0.75*n.distance:
                    good += 1

            # Use max for similarity
            sift_similarity[i,j] = max(good/len(kp1), good/len(kp2))
            sift_similarity[j,i] = sift_similarity[i,j]

    # Put photo id to array and save as csv
    photofile_list = [int(x[:-4]) for x in photofile_list]
    sift_similarity = np.insert(sift_similarity, 0, photofile_list, axis=0)
    np.savetxt(f'sift/{cluster_id}.csv', sift_similarity, delimiter=',')
```

opencv의 sift를 사용해서 구현했다. matches의 distance에 ratio test를 적용해서 잘 매칭된 matches를 뽑고 이를 전체 key point의 개수로 나눠 similarity score을 측정한다. ratio와 similarity score 방식은 논문을 참고했다.¹⁾

graph의 similarity threshold를 추정하기 위해 유사한 이미지로 채워진 cluster : 8에 대해 SIFT algorithm을 적용했다. 순서대로 1~8번 사진이라 하자.



다음은 pairwise similarity 표이다.

	1	2	3	4	5	6	7	8
1	0	0.0284280936	0.00877926421	0.0284280936	0.00717964726	0.0104485678	0.216292135	0.014632107
2	0.0284280936	0	0.674157303	0.495767836	0.00109255502	0.00144118177	0.00280898876	0.612359551
3	0.00877926421	0.674157303	0	0.530834341	0.000780396441	0.00144118177	0.00200642055	0.64756447
4	0.0284280936	0.495767836	0.530834341	0	0.00124863431	0.00198162493	0.00160513644	0.431680774
5	0.00717964726	0.00109255502	0.000780396441	0.00124863431	0	0.0294998855	0.0229436554	0.023255814
6	0.0104485678	0.00144118177	0.00144118177	0.00198162493	0.0294998855	0	0.0381913169	0.0190956584
7	0.216292135	0.00280898876	0.00200642055	0.00160513644	0.0229436554	0.0381913169	0	0.0116372392
8	0.014632107	0.612359551	0.64756447	0.431680774	0.023255814	0.0190956584	0.0116372392	0

상당히 유사해보이는 2~4사이의 similarity가 0.49~0.67로 생각했던 것보다 훨씬 낮았다. 또한 6, 7번 사진의 경우 상당히 유사해 보임에도 similarity는 0.038로 매우 낮았다. 이 상황에서 threshold를 적용하는 경우 같은 cluster에 묶여야 하는 사진도 나뉠 가능성이 매우 높았다.

```
def sift(cluster_id):
    dir = f'geo_cluster/{cluster_id}'
    photofile_list = os.listdir(dir)
    sift_similarity = np.zeros((len(photofile_list), len(photofile_list)))

    for i in range(len(photofile_list)-1):
        for j in range(i+1, len(photofile_list)):
            img1 = cv2.imread(dir + '/' + photofile_list[i])
            img2 = cv2.imread(dir + '/' + photofile_list[j])
            gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
            gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)

            # SIFT detector
            detector = cv2.xfeatures2d.SIFT_create()
            kp1, desc1 = detector.detectAndCompute(gray1, None)
            kp2, desc2 = detector.detectAndCompute(gray2, None)

            # BFMatcher
            matcher = cv2.BFMatcher(cv2.NORM_L2, crossCheck=True)
            # calculate matching
            matches = matcher.match(desc1, desc2)
            # Use max for similarity
            sift_similarity[i,j] = max(len(matches)/len(kp1), len(matches)/len(kp2))
            sift_similarity[j,i] = sift_similarity[i,j]

    # Put photo id to array and save as csv
    photofile_list = [int(x[:-4]) for x in photofile_list]
    sift_similarity = np.insert(sift_similarity, 0, photofile_list, axis=0)
    np.savetxt(f'sift/{cluster_id}.csv', sift_similarity, delimiter=',')
```

similarity를 구하는 다른 방식을 고안하였다. key points에서 잘 match된 point를 찾고 그 개수 비율로 similarity를 구하는 방식이다.

	1	2	3	4	5	6	7	8
1	0	0.536516854	0.53008596	0.481257557	0.434782609	0.404682274	0.444816054	0.558064516
2	0.536516854	0	0.720630372	0.685393258	0.674157303	0.596910112	0.519662921	0.759677419
3	0.53008596	0.720630372	0	0.70773639	0.653295129	0.616045845	0.521489971	0.761290323
4	0.481257557	0.685393258	0.70773639	0	0.634824667	0.591293833	0.480048368	0.7
5	0.434782609	0.674157303	0.653295129	0.634824667	0	0.26715907	0.411717496	0.677419355
6	0.404682274	0.596910112	0.616045845	0.591293833	0.26715907	0	0.394462279	0.638709677
7	0.444816054	0.519662921	0.521489971	0.480048368	0.411717496	0.394462279	0	0.551612903
8	0.558064516	0.759677419	0.761290323	0.7	0.677419355	0.638709677	0.551612903	0

그 결과 다음과 같은 합리적인 similarity를 구할 수 있었다.

```

def flood_fill(cluster_id, threshold):
    # read sift result from csv file
    filename = f'sift/{cluster_id}.csv'
    dataset = np.loadtxt(filename, delimiter=',', dtype=np.float64)

    # data process to flood fill algorithm
    photo_id = dataset[0, :].astype(dtype=np.int64)
    num_photo = photo_id.shape[0]
    similarity_arr = dataset[1:, :]
    connected_graph = np.where(similarity_arr > threshold, 1, 0)
    label = np.zeros(photo_id.shape).astype(dtype=np.int64)
    label.fill(-1)

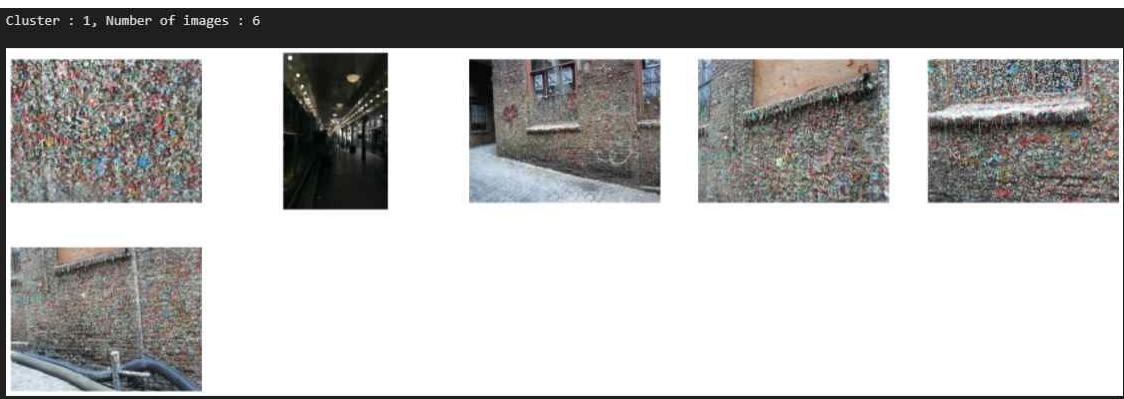
    labeling_index = 0
    stack = []
    # Use dfs
    for i in range(num_photo):
        if label[i] != -1:
            continue
        stack.append(i)
        while(stack):
            node = stack.pop()
            # pass if labeled
            if label[node] != -1:
                continue
            label[node] = labeling_index
            for index, connect in enumerate(connected_graph[node, :]):
                if connect == 1:
                    stack.append(index)
        labeling_index += 1

    photo_cluster = np.column_stack((photo_id, label))
    for i in range(labeling_index):
        photo_list = photo_cluster[photo_cluster[:, 1]==i][:, 0]
        print(f'Cluster : {i}, Number of images : {len(photo_list)}')
        fig = plt.figure(figsize=(30, 20))
        rows = 4
        columns = 5
        if len(photo_list) >=20:
            sample_img = np.random.choice(photo_list, 20, replace=False)
        else:
            sample_img = np.random.choice(photo_list, len(photo_list), replace=False)
        for j in range(len(sample_img)):
            tmp_img = cv.imread(f'./geo_cluster/{cluster_id}/{sample_img[j]}.jpg')
            fig.add_subplot(rows, columns, j+1)
            plt.axis('off')
            plt.imshow(cv.cvtColor(tmp_img, cv.COLOR_BGR2RGB))
        plt.show()

```

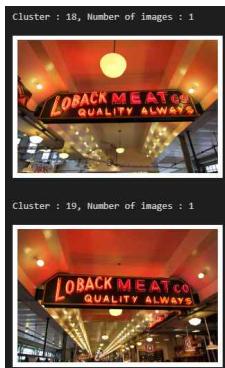
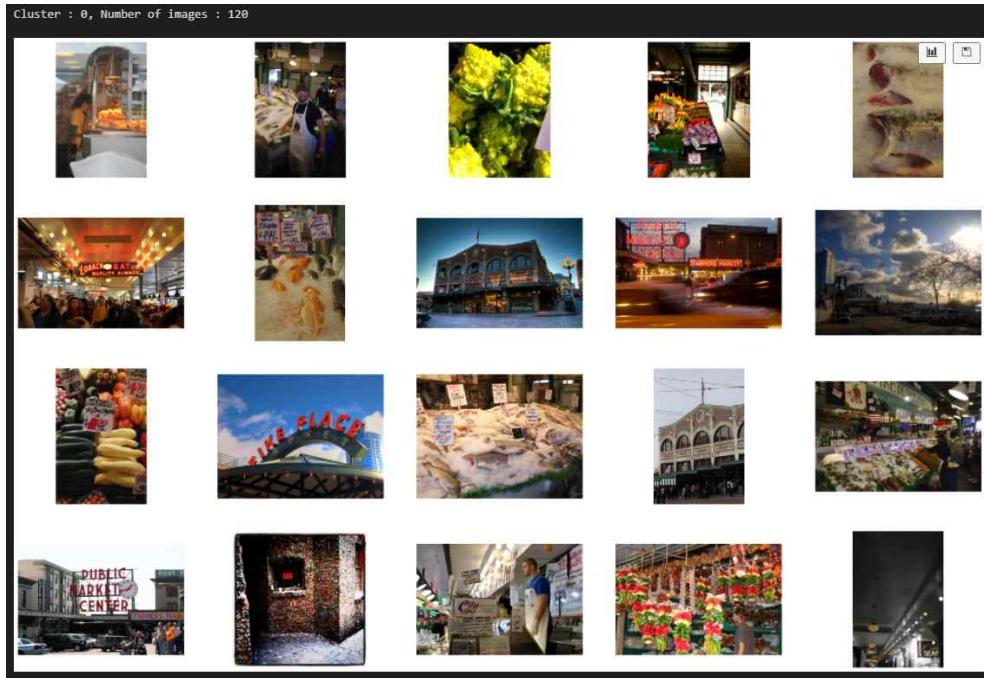
위에서 구한 similarity를 바탕으로 Flood fill 알고리즘을 통해 clustering하였다. threshold를 넘는 similarity를 갖는 요소끼리 연결된 graph로 간주하고 연결된 graph끼리 labeling을 하였다. graph 탐색 시 dfs를 사용하였다. 마찬가지로 labeling된 cluster마다 random sample 사진을 통해 clustering 결과를 확인했다.

4. 결과 및 분석



특정 경우에 이처럼 비교적 잘 clustering 되었다고 볼 수 있었다.

cluster : 0_0_0_0



하지만 많은 경우 다음과 같이 비슷하지 않은 사진들이 같이 clustering되고 비슷한 cluster는 따로 분류되었다. threshold를 변경해보면서 시도해 보았지만 similarity 값 자체가 현실과 거리가 있었다. SIFT를 통한 visual clustering은 특수한 상황에만 사용할 수 있었고, 범용적이지 못했다.

<Phase 2-2 : Visual clustering with Geographical distance>

1. 개요

SIFT의 similarity 정확도와 신뢰도가 높지 않은 상황에서 이를 보완하기 위해 기존에 clustering이 잘 진행되었던 Geographical feature을 같이 사용했다. Euclidean distance를 구하고 distance가 작을수록 높은 score을 부여해서 이를 SIFT similarity와 합쳐서 새로운 score을 구했다. 계수를 통해 distance feature과 visual feature을 조정했다. Multi feature을 바탕으로 Visual feature만 사용했을 때보다 더 나은 결과를 확인할 수 있었다.

2. 구현

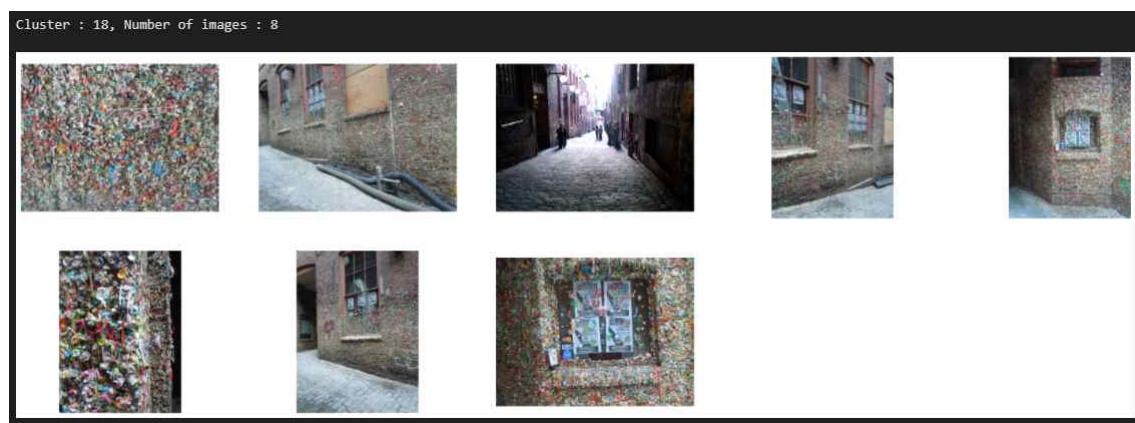
```
distance_list = np.zeros((num_photo, num_photo))
distance_list = distance_list.astype(np.float64)
for i in range(num_photo-1):
    for j in range(i+1, num_photo):
        i_lat = geo_dataset[geo_dataset['photo_id'] == photo_id[i]]['latitude'].iloc[0]
        j_lat = geo_dataset[geo_dataset['photo_id'] == photo_id[j]]['latitude'].iloc[0]
        i_long = geo_dataset[geo_dataset['photo_id'] == photo_id[i]]['longitude'].iloc[0]
        j_long = geo_dataset[geo_dataset['photo_id'] == photo_id[j]]['longitude'].iloc[0]
        distance_list[i, j] = 1/((i_lat - j_lat)**2 + (i_long - j_long)**2)
        distance_list[j, i] = distance_list[i, j]
```

거리가 가까울수록 높은 score을 주기 위해 $\frac{1}{distance}$ 를 사용했다. 이 경우에 아주 가까운 거리에서 찍었을 경우 score가 ∞ 에 가깝게 발산하는 point set이 존재했다. 이에 따라 Normalize를 하면 나머지 값이 전부 0이 되었다.

```
distance_list = np.zeros((num_photo, num_photo))
distance_list = distance_list.astype(np.float64)
for i in range(num_photo-1):
    for j in range(i+1, num_photo):
        i_lat = geo_dataset[geo_dataset['photo_id'] == photo_id[i]]['latitude'].iloc[0]
        j_lat = geo_dataset[geo_dataset['photo_id'] == photo_id[j]]['latitude'].iloc[0]
        i_long = geo_dataset[geo_dataset['photo_id'] == photo_id[i]]['longitude'].iloc[0]
        j_long = geo_dataset[geo_dataset['photo_id'] == photo_id[j]]['longitude'].iloc[0]
        distance_list[i, j] = ((i_lat - j_lat)**2 + (i_long - j_long)**2)
        distance_list[j, i] = distance_list[i, j]
tmp_distance_list = np.max(distance_list) - distance_list
norm_distance_inverse_list = (tmp_distance_list - np.min(tmp_distance_list))/(np.max(tmp_distance_list)-np.min(tmp_distance_list))
similarity_arr = dataset[1:, :]
geo_similar_arr = k * norm_distance_inverse_list + (k-1) * similarity_arr
connected_graph = np.where(geo_similar_arr > threshold, 1, 0)
```

위에서 발생한 문제를 해결하기 위해 distance의 max값에 전체 numpy array를 빼서 $y = b - x$ 형태로 변환했다. 여기에 min-max Normalization을 적용해 [0, 1]의 범위에 distance score을 할당했다. 이 distance score와 기존 SIFT similarity score을 k라는 계수를 통해 더한 후 threshold를 적용해 connectivity를 구했다. 나머지 과정은 위 Flood-fill 알고리즘과 동일하다.

3. 결과 및 분석



이전에 비해 더 나은 clustering 결과를 확인할 수 있었다. 하지만 이렇게까지 clustering하기 위해서 parameter tuning을 오래 거쳐야 했다. 잘 묶이지 않는 개별 cluster들이 많아졌다.

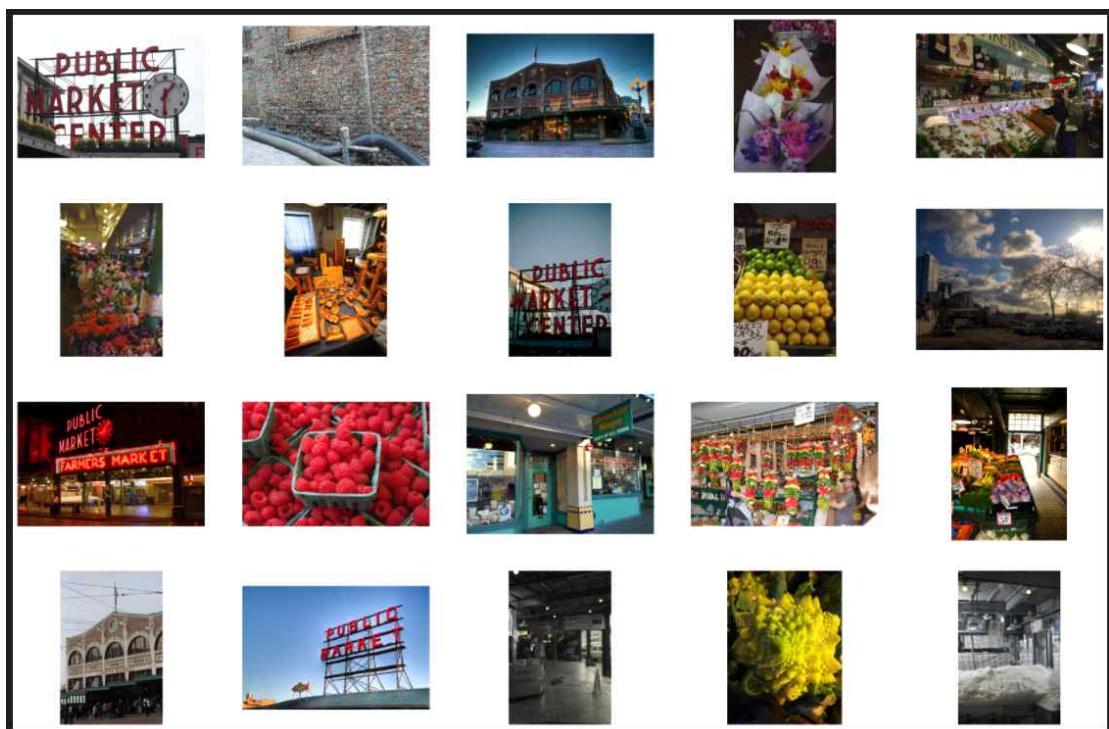
<Phase 3 : Textual clustering>

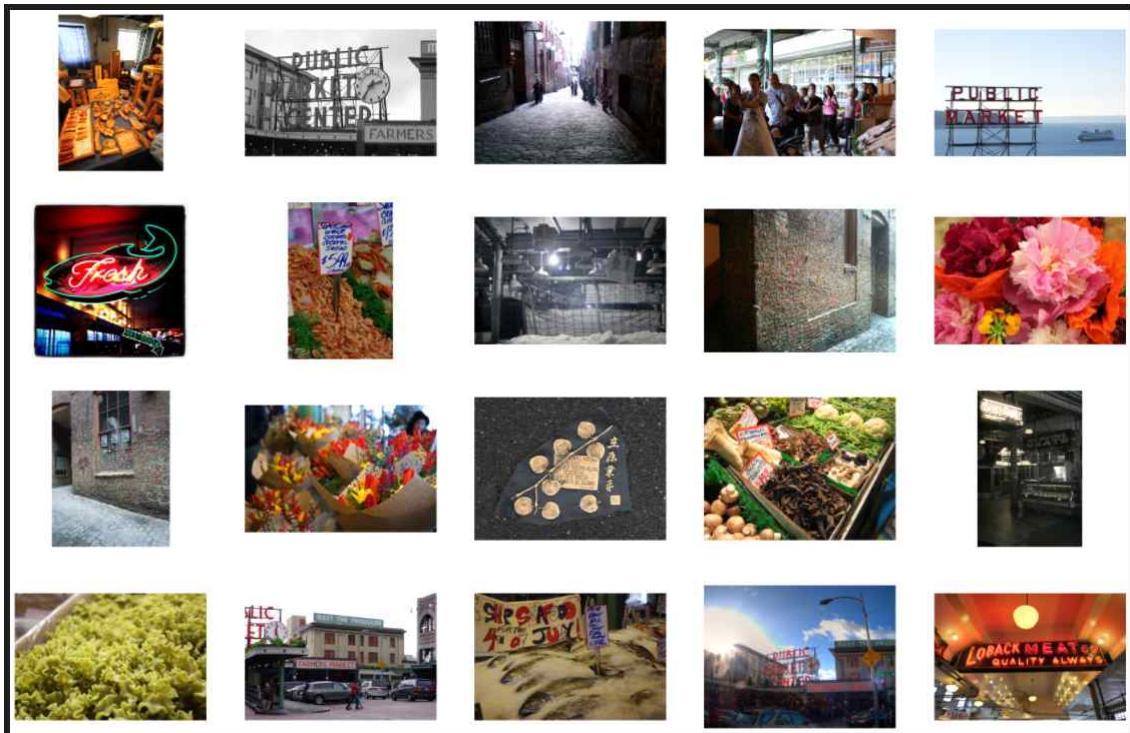
1. 개요

시도 1) Word2Vec을 이용한 landmark tag 추출

1	seattle	7563
2	washington	2958
3	usa	785
4	wa	718
5	convention	667
6	2009	533
7	square	512
8	redmond	509
9	2012	501
10	iphoneography	492
11	uploaded:by=instagram	489
12	squareformat	481
13	instagramapp	478
14	park	474
15	rf	444
16	unitedstates	431
17	2010	411
18	furry	402
19	con	402
20	rainfurrest	402
21	rf2012	402
22	eyefi	378
23	art	372
24	kinner	339
25	concerts	336
26	naticityrollergirls	334
27	rcng	333
28	rollerderby	333
29	2004	333
30	lakeunion	322

Tag.csv에 어떤 tag들이 있는지 파악하고자 tag들을 빈도수의 내림차순으로 정렬했다. 그 결과 seattle, washington과 같은 지역명이 상위권에 포진되어 있음을 확인할 수 있다. 이러한 Tag가 왜 있는지 생각해보다 이러한 Tag는 지역 주민이 아닌 외부 관광객이 붙였을 가능성이 높다고 판단했고, 관광객은 지역 주민보다 landmark와 attraction을 찍었을 가능성이 높다고 생각했다. 이에 seattle이라는 tag를 가진 photo를 랜덤하게 나열해보았다.





그 결과 다른 cluster보다 landmark와 attraction의 비중이 매우 높음을 확인했다. 원래라면 Visual clustering에 의해 나뉘어진 같은 landmark들을 유사도 체크를 통해 merge해야 하고 그 과정에서 seattle과 같은 지명은 stop word로 meaningless하게 처리해야 했다. 하지만 SIFT으로 인한 visual clustering의 정확도가 낮은 상황에서 이러한 cluster들을 text clustering 하게 될 경우 하나의 큰 cluster로 merge될 가능성이 높았다. 이에 따라 tag 정보로 섞여 있는 landmark를 한 번 더 구분한 다음 merge를 시도했다.

```

photo_id_tag_list = {}
for index, row in tags.iterrows():
    photo_id = row['photo_id']
    tag = str(row['tag'])
    # if tag contain = : or only with number, delete
    if '=' in tag:
        continue
    if ':' in tag:
        continue
    if tag.isdigit():
        continue

    if photo_id in photo_id_tag_list:
        photo_id_tag_list[photo_id].append(tag)
    else:
        photo_id_tag_list[photo_id] = [tag]

tag_sentence_list = []
for key, value in photo_id_tag_list.items():
    tag_sentence_list.append(' '.join(value))

normalized_text = []
for string in tag_sentence_list:
    tokens = string.lower()
    normalized_text.append(tokens)

tag_sentence = [word_tokenize(sentence) for sentence in normalized_text]

model = Word2Vec(sentences=tag_sentence, window=1000000, min_count=3, workers=12, sg=0)

```

Word2Vec을 이용해 model을 구현했다. photo_id_tag_list라는 딕셔너리에 photo id를 key로 tag들을 리스트 형식으로 append 했다. 이때 tag에 =, :, 년도와 같이 불필요한 정보가 있는 tag는 추가하지 않았다. 이렇게 나온 각 photo id별 tag list를 하나의 sentence로 만들어 tag_sentence_list에 저장했다. 그 후 normalize와 tokenize 과정을 거친 후 모델에 대입했다. 이때 완성된 문장이 아니므로 window의 크기는 최대한 크게 설정했고, sg=0을 통해 CBOW method를 사용했다. 이렇게 완성된 model에 seattle을 넣어 similarity를 얻었고, 그 결과는 다음과 같다.

```
[('columbia', 0.9280167818069458), ('greatwheel', 0.9221882820129395), ('pikeplace', 0.9217881560325623), ('design', 0.921519935131073), ('pikeplacemarket', 0.9195425510406494), ('place', 0.918896496295929), ('neon', 0.9184477925300598), ('northwest', 0.9171898365020752), ('publicmarket', 0.9167032241821289), ('washingtonstate', 0.915571391582489)]
```

greatwheel이나 pikeplace, pikeplacemarket, publicmarket 등 landmark에 가까운 tag들이 나왔지만, 중간에 다른 정보도 섞여있었고, 이를 통해 모든 landmark tag를 커버할 수는 없었다. word embedding 모델 특성상 단어 간 유사도가 대체될 수 있는지를 나타내는 경향이 있어서 같이 쓰인 단어 셋을 내보내진 않았다고 추측한다.

시도 2) pairwise similarity check

```
def tag_clustering(cluster_id):
    path_photos = f'./geo_cluster/{cluster_id}/'
    photofile_list = os.listdir(path_photos)
    photo_id_list = []
    for photofile in photofile_list:
        photo_id_list.append(photofile[:-4])
    num_photos = len(photo_id_list)

    tag_similar = np.zeros((num_photos, num_photos))

    for i in range(num_photos-1):
        for j in range(i+1, num_photos):
            if int(photo_id_list[i]) in photo_id_tag_list and int(photo_id_list[j]) in photo_id_tag_list:
                tag_similar[i, j] = model.wv.n_similarity(photo_id_tag_list[int(photo_id_list[i])], photo_id_tag_list[int(photo_id_list[j])])
                tag_similar[j, i] = tag_similar[i, j]

    np.savetxt(f'tag_result/{cluster_id}.csv', tag_similar, delimiter=',')
tag_clustering('0_0_0_0')
```

위에서 구현한 모델에서 seattle, washington, instagramapp과 같은 단어를 stop word로 설정하고 Word2Vec 모델을 학습시켰다. 그리고 그 model에 대해 wv.n_similarity()라는 함수를 사용해 사진간 tag set의 유사성을 계산했다. wv.n_similarity()는 word로 이루어진 list를 2개를 받고, 그 word set 간의 유사성을 계산하는 함수이다. tag가 있는 사진에 한 해 pairwise하게 similarity를 구하고 csv파일로 저장했다. 이 pairwise한 similarity array를 <Phase 2-2>에서 진행한 것과 같이 geo feature, visual feature, text feature간의 similarity를 계수를 통해 더해서 구했다. 하지만 이 경우에는 tag가 없는 사진 set이 많았고, 이에 따라 tag가 없는 사진은 가까운 거리, 높은 visual similarity를 가지고 있음에도 threshold를 통과하지 못했다.

시도 3) merge small cluster by text clustering and clustering with geographical feature

- <Phase 2-2>의 결과로 얻은 cluster는 2가지 문제점이 있었다.
 - (1) 같은 cluster내에 다른 landmark가 섞여있는 경우가 존재한다.
 - (2) 다른 cluster에 같은 landmark가 있는 경우가 존재한다.

이를 해결하기 위해 text clustering을 진행해서 naive하게 merge한 다음 그 내에서 다시 geographical clustering을 적용해서 분류한다. 그렇게 되면 (2)번 case에 해당하는 cluster 들 중에 tag가 있는 landmark는 merge될 것이다. (1)번 case 또한 분리되지 않은 landmark를 다른 cluster와 합쳐서 clustering하게 될 경우 밀집된 point가 늘어나 더 잘 clustering 될 수 있을 것이다. 또한 tag의 similarity가 높아 merge된 museum과 같은 경우 geographical하게 분리해야 서로 다른 landmark로 clustering 할 수 있다.

2. 구현

```

tag_list_1 = {}
for id in photo_id_list_1:
    if int(id) in photo_id_tag_list:
        for tmp_tag in photo_id_tag_list[int(id)]:
            if tmp_tag in tag_list_1:
                tag_list_1[tmp_tag] += 1
            else:
                tag_list_1[tmp_tag] = 0

tag_list_2 = {}
for id in photo_id_list_2:
    if int(id) in photo_id_tag_list:
        for tmp_tag in photo_id_tag_list[int(id)]:
            if tmp_tag in tag_list_2:
                tag_list_2[tmp_tag] += 1
            else:
                tag_list_2[tmp_tag] = 0

sorted_1 = sorted(tag_list_1.items(), key=lambda x: x[1], reverse=True)
sorted_2 = sorted(tag_list_2.items(), key=lambda x: x[1], reverse=True)
try:
    similarity = model.wv.similarity(sorted_1[0][0], sorted_2[0][0])
except:
    similarity = 0
return similarity

```

처음에는 DTM을 이용해 가장 많이 사용된 tag끼리 wv.similarity() 함수를 이용해 similarity를 계산했다. 이렇게 계산하니 tag가 많이 쓰이지 않아 model bag of word에서 탈락된 작은 cluster에 대해 오류를 일으켰다. 그러한 경우에 try, except 문을 통해 계산했으나, threshold를 작게 해도 merge 되지 않는 작은 cluster들이 많았다.

```

tag_list_1 = []
for id in photo_id_list_1:
    if int(id) in photo_id_tag_list:
        tag_list_1.extend(photo_id_tag_list[int(id)])

tag_list_2 = []
for id in photo_id_list_2:
    if int(id) in photo_id_tag_list:
        tag_list_2.extend(photo_id_tag_list[int(id)])
similarity = model.wv.n_similarity(tag_list_1, tag_list_2)
return similarity

```

다음으로 cluster 내의 모든 tag를 이어 붙인 list 2개를 wv.n_similarity() 함수를 이용해 similarity를 구했다. 적절한 threshold를 설정해 cluster를 merge 할 수 있었다.

```

def text_clustering(threshold):
    path = f'./final_cluster/'
    cluster_ids = os.listdir(path)
    num_cluster = len(cluster_ids)
    cluster_merge = np.zeros((num_cluster, num_cluster))
    for i in range(num_cluster-1):
        for j in range(i+1, num_cluster):
            cluster_merge[i, j] = text_similarity(cluster_ids[i], cluster_ids[j])
            cluster_merge[j, i] = cluster_merge[i, j]

    connected_graph = np.where(cluster_merge > threshold, 1, 0)
    label = np.zeros(num_cluster).astype(dtype=np.int64)
    label.fill(-1)

    labeling_index = 0
    stack = []
    # Use dfs
    for i in range(num_cluster):
        if label[i] != -1:
            continue
        stack.append(i)
        while(stack):
            node = stack.pop()
            # pass if labeled
            if label[node] != -1:
                continue
            label[node] = labeling_index
            for index, connect in enumerate(connected_graph[node, :]):
                if connect == 1:
                    stack.append(index)
        labeling_index += 1
    merged_cluster = np.column_stack((cluster_ids, label))

    for i in range(labeling_index):
        merged_list = merged_cluster[merged_cluster[:, 1]==str(i)][:, 0]
        print(f'Cluster : {i}, Number of images : {len(merged_list)}')
        print(merged_list)

    return merged_cluster, labeling_index

```

<Phase 2-2>의 결과로 나온 모든 cluster가 담겨있는 폴더를 탐색해 모든 cluster에 대해 pairwise한 similarity를 구한다. (#cluster x #cluster)의 array에 similarity를 저장하고 threshold를 적용해 connectivity를 체크한다. 연결된 cluster끼리 Flood-fill 알고리즘을 적용해 라벨링을 한다. 이때 위에서와 마찬가지로 stack을 이용한 dfs를 적용한다. 이후 연결된 graph끼리 merge한다.

```

def result_geo(cluster_id):
    photos = pd.read_csv('./Photo.csv', header=0, names= ['photo_id', 'user_id', 'latitude', 'longitude', 'time', 'etc'])
    dataframe = pd.DataFrame(photos)
    photo_id_list = []
    path_photos = f'./result_cluster/{cluster_id}/'
    photofile_list = os.listdir(path_photos)
    for photofile in photofile_list:
        photo_id_list.append(photofile[:-4])

    result_dataframe = pd.DataFrame()
    for id in photo_id_list:
        result_dataframe = pd.concat([result_dataframe, dataframes[dataframe['photo_id'] == int(id)][['latitude', 'longitude']]])

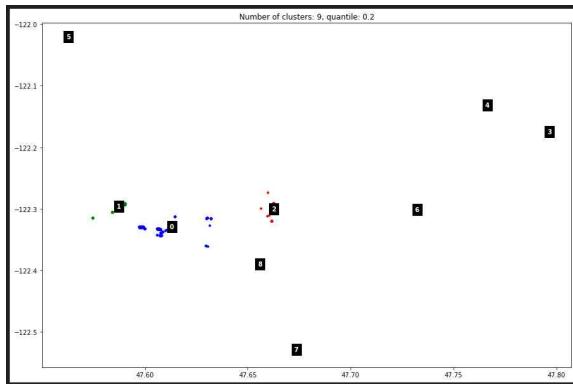
```

Textual clustering 한 결과 중 geo-clustering이 필요하다 판단되는 set에 한해 geo-clustering을 하였다. clustering된 사진 폴더에서 사진 list를 불러와 Photos.csv와 매칭한 후 위도, 경도 값을 얻었다.

4. 결과 및 분석

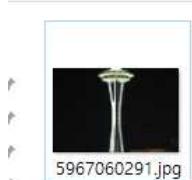
```
Cluster : 0, Number of images : 4
['0_0_0_0' '0_0_0_11' '0_0_0_3' '0_0_2_3']
Cluster : 1, Number of images : 1
['0_0_0_1']
Cluster : 2, Number of images : 1
['0_0_0_13']
Cluster : 3, Number of images : 3
['0_0_0_2' '0_0_1_2' '6_9']
Cluster : 4, Number of images : 25
['0_0_0_4' '0_0_0_9' '0_0_11' '0_0_1_0' '0_0_1_4' '0_0_21' '0_0_2_1'
 '0_0_2_4' '0_0_6' '0_10' '0_1_5' '0_3_3' '0_3_6' '0_5_2' '0_5_5' '0_6_10'
 '0_9' '11_1' '16' '1_2' '1_3' '5_4' '7_10' '7_11' '8']
Cluster : 5, Number of images : 1
['0_0_0_6']
Cluster : 6, Number of images : 5
['0_0_0_7' '0_0_7' '0_1_3' '0_6_0' '1_5']
Cluster : 7, Number of images : 1
['0_0_0_8']
Cluster : 8, Number of images : 1
['0_0_10']
Cluster : 9, Number of images : 1
['0_0_17']
Cluster : 10, Number of images : 2
['0_0_1_1' '0_0_2_6']
Cluster : 11, Number of images : 1
...
Cluster : 48, Number of images : 1
['6_0']
Cluster : 49, Number of images : 1
['6_3']
```

Textual clustering을 통해 182개의 cluster가 50개의 cluster로 감소했다.



특정 cluster에서는 index들이 강하게 뭉쳐있는 좌표가 있어서 geo-clustering의 효과가 좋았다. 위 사진은 182개의 사진을 가지고 있는 cluster에서 geo clustering한 예시이다. 이 과정을 통해 50개의 cluster가 57개로 늘어났다.

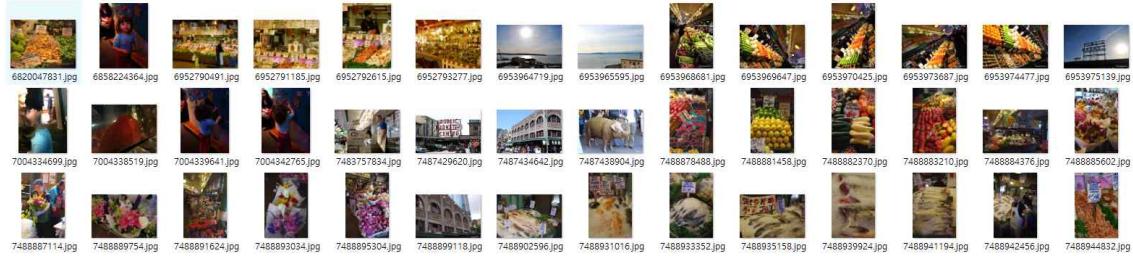
ta_project > result_cluster > new20



반면 space needle tower와 같이 다양한 위치에서 찍힐 수 있는 landmark의 경우 이런 식으로 동떨어진 outlier cluster 가 발생할 수 있다.

cluster들의 예시이다.

case 1) cluster : new0

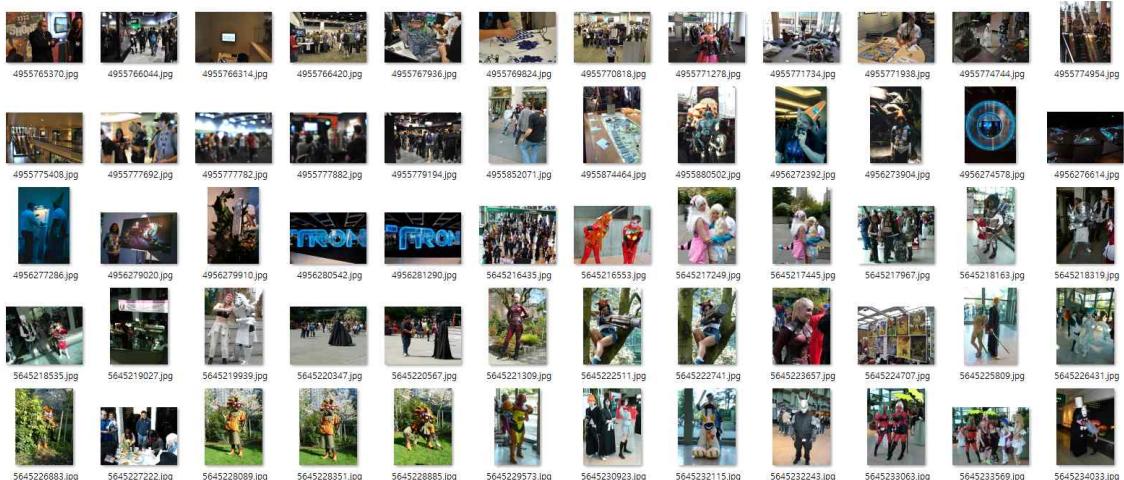


이 경우 market과 유사한 tag를 가진 cluster들이 합쳐졌다. 반복적인 geo-clustering으로 인해 market sign이 여러 cluster로 나뉘어져 있었는데, Textual clustering을 통해 한 cluster로 합칠 수 있었다. 그 과정에서 Gum house가 포함되었는데, 이는 gum tag 자체가 market과 similarity가 커서 merge되었다. 연관된 Tag는 pikeplace, marketplace 등이다.



이 두 사진은 textual, geographical similarity는 낮았지만, Visual similarity가 높아 <Phase 2-2>에서 같은 cluster에 묶였고, <Phase 3>에 따라 같이 merge 된 것으로 추측한다.

case 2) cluster : new10



다른 cluster에 흩어져있던 cosplay 사진들이 Textual clustering을 통해 한 cluster로 합쳐졌다. 다른 이미지도 포함되어 있었지만 비교적 잘 clustering 되었다. 연관된 Tag는 sakuracon, comic, cosplay 등 이었다.

case 3) cluster : new2



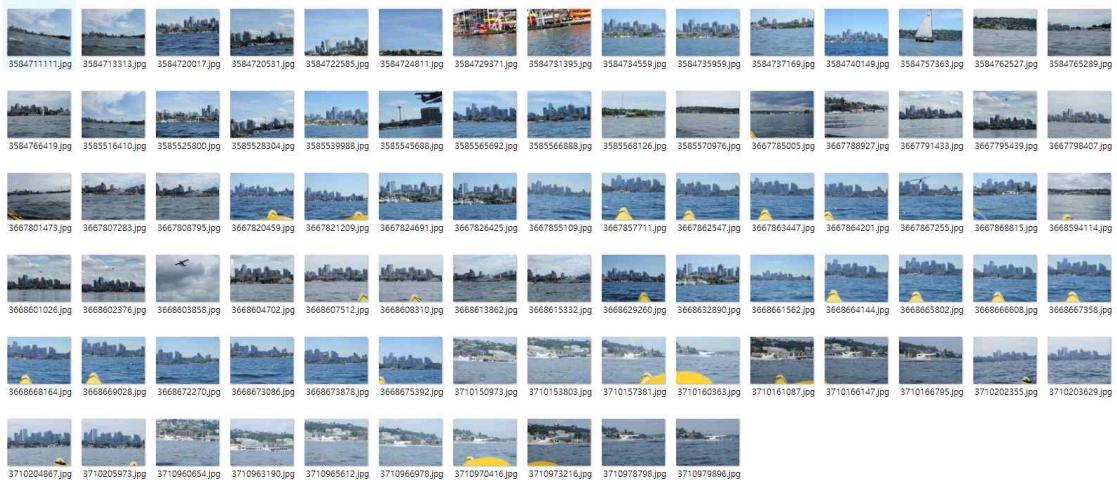
<Phase 2-2>에 의해 잘게 나뉜 cluster 중에 Tag를 가지고 있지 않아 merge에 누락된 사례도 존재했다.

case 4) cluster : new15



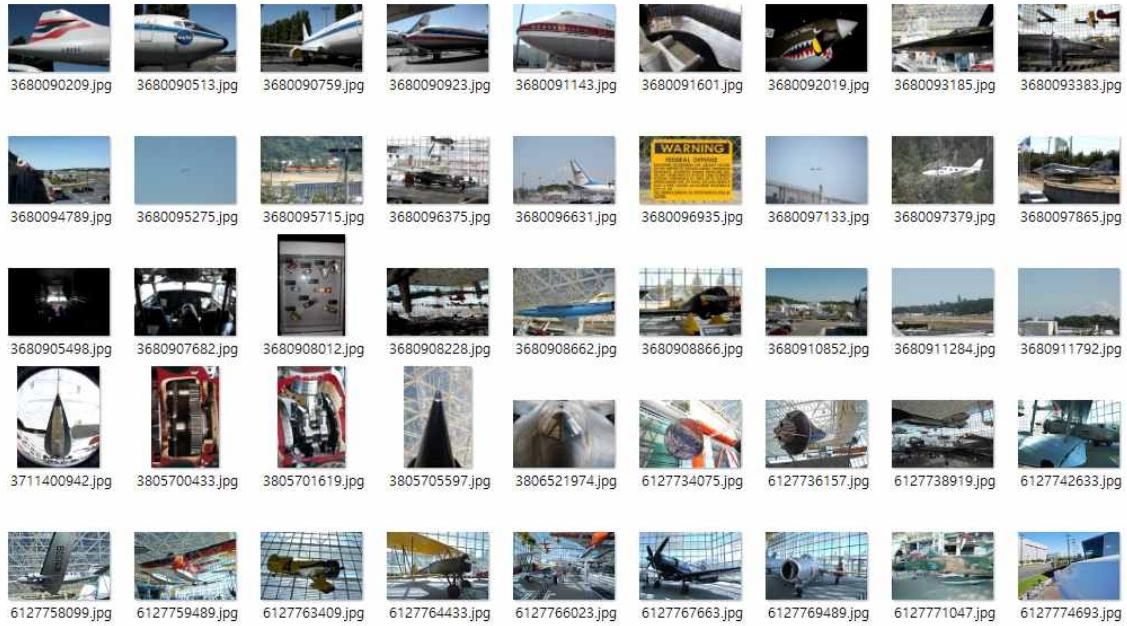
축구장이 잘 clustering 됐음을 확인할 수 있다. 이 경우에는 Textual clustering에 의해 merge되지 않았음에도 처음부터 잘 clustering 된 사례이다. 관련 Tag는 soccer, sounders, field, football이다.

case 5) cluster : new37



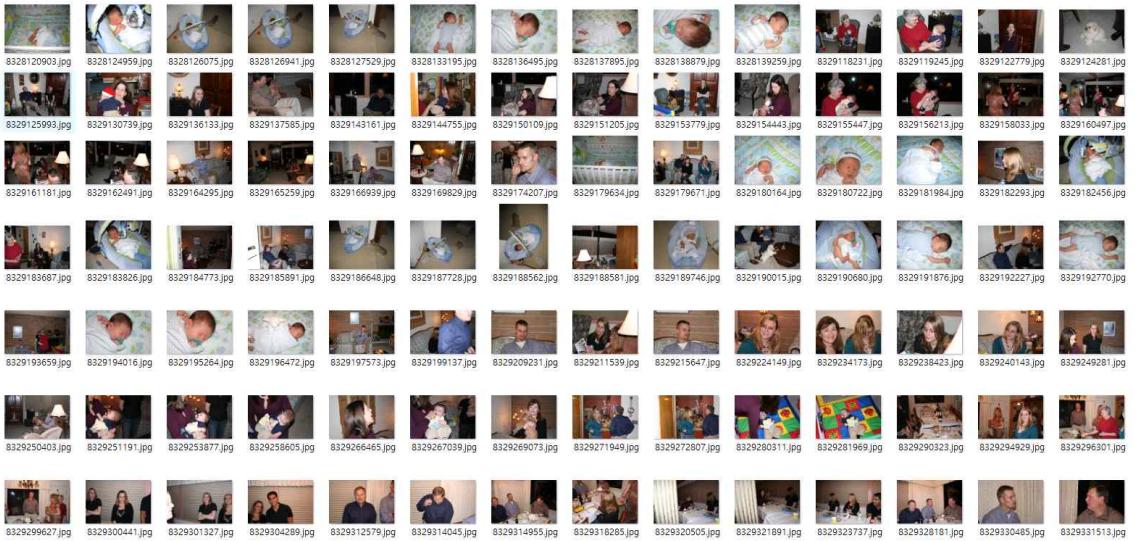
Textual clustering에 의해 잘 cluster된 사례이다. 기존에 나뉘어져 있던 cluster가 잘 merge 되었다. outlier가 된 경우가 있지만, 전체 비율을 생각한다면 잘 clustering 되었다고 볼 수 있다. 관련된 Tag는 lakeunion, kayak, seaplane 등이다.

case 6) cluster : new42



이것도 마찬가지로 geo clustering 단계부터 잘 clustering 되었던 경우이다. 동떨어진 위치에 point가 밀집되어 있어서 그런 것으로 추정한다. 관련된 Tag는 plane, c130 등이다.

case 7) cluster : new25



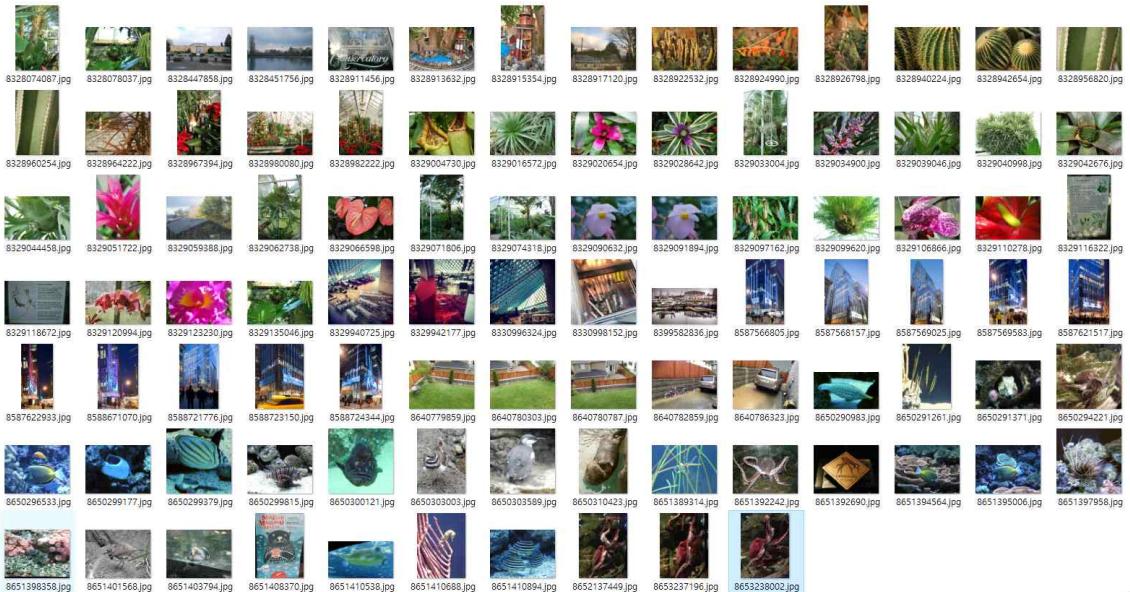
landmark는 아니지만 recursive하게 clustering 했을 때 잘 분류되었던 cluster이다. 이러한 landmark가 아닌 cluster도 많았다. 이를 Word2Vec을 이용해서 seattle과 관련된 landmark tag를 통해 분류하려 했으나, 실패했다.

case 8) cluster : new26



seattle의 가장 유명한 landmark라고 할 수 있는 space needle tower는 시애틀 전역에서 볼 수 있기 때문에 point가 밀집되지 않았다. 이에 따라 각기 다른 cluster에 분류되었고, 이를 Textual clustering을 통해 merge하다 보니 다른 이미지가 많이 유입되었다. 이렇게 큰 landmark는 geo clustering을 하기 전에 tag를 통해 먼저 제거하고 clustering한 다음 나중에 합쳤으면 더 좋은 결과가 있었을 것으로 추측한다.

case 9) cluster : new4



위와 같이 다양한 사진이 혼합된 cluster도 존재했다. 이를 geographical하게 분류하려 했으나 크기가 1인 cluster만 많아지고 정작 landmark들은 잘 분류되지 않았다. Feature을 다각화 했다면 좀 더 정밀한 분류가 가능했을 것으로 추측한다.

<Overall review>

생각보다 recursive하게 geo clustering 하는 것이 잘 작동했다. clustering 개수가 많이 늘어났지만, landmark 별로 분류를 하기 위해서는 trade-off라 생각한다. 대다수의 경우에 qunatile 값은 0.1을 사용하였다. qunatile 값을 변경해가면서 실험해봤을 때 0.1이 적절한 결과를 도출했다. qunatile이 커지면 cluster의 크기가 커지고 개수가 작아졌고, qunatile이 작아지면 cluster의 크기가 작아지고 개수가 많아졌다. 좀 더 작고 세밀하게 나눠야 하는 경우 0.05를 사용했고 크고 범용적으로 나눠야 하는 경우 0.15를 사용하였다.

Visual clustering을 할 때 SIFT similarity score 방식이 아쉬웠다. 다양한 방식을 시도해 보았으나 similarity score가 현실과 거리감이 있었다. 만약 similarity score을 잘 계산해서 visual clustering이 잘 이루어졌으면 더 좋은 결과가 있었을 것이다. 또한 SURF 알고리즘을 사용하지 못한 점이 아쉬웠다. opencv의 cuda 세팅을 완료했으나, SIFT는 cuda를 제공하지 않았고, SURF는 제공했지만 유료 라이센스였다. SURF 자체가 SIFT에 비해 약 3배 정도 빠르다고 알려져 있었으므로 cuda까지 이용했다면 더 빠른 시간에 visual similarity를 계산할 수 있었을 것이다. 그렇다면 Geographical clustering이 잘 실행되었는지에 대한 score로 다른 cluster와의 visual similarity를 이용해 stop point를 잘 결정할 수 있었을 것이다. visual similarity에 대한 신뢰도가 낮을 때 다른 feature과 조합해서 해결할 수 있다는 insight를 얻었다.

<Phase 3>의 시도 2에서 만약 모든 photo에 대해 tag가 존재했다면 textual, geographical, visual feature을 모두 한꺼번에 고려해서 clustering 할 수 있었을 것이다. space needle tower와 같이 거대한 landmark의 경우 산발적인 장소에서 사진이 찍힐 가능성이 높으므로 tag clustering을 통해 data set에서 제외한 다음 geo clustering 했으면 더 성능이 좋았을 것이다. 특정 지명에 대한 Word2Vec similarity를 이용해서 landmark tag를 결정할 수 있는 가능성을 보았다. 실제로 ‘seattle’에 대해 높은 similarity를 가지고 있는 tag는 landmark이었던 경우가 많음을 확인했다.

이번 task에 대한 전체적인 구조는 hierarchical clustering이었다. 그중에서도 top-down method였다. hierarchical clustering에 대해 bottom-up method가 clustering 성능이 좋다고 알려져 있다.²⁾ distance feature과 visual similarity, textual similarity를 적절하게 조합해서 overall similarity를 구하고 이를 통해 bottom-up method을 사용했다면 더 좋은 결과가 있었을 것으로 추측한다.

다양한 feature에 대한 clustering 기법을 조합해서 한 가지를 사용했을 때보다 정확도를 높일 수 있음을 확인했다. 시간이나 user id와 같은 다른 feature를 시도했다면 더 좋은 결과가 있었을 것으로 추측한다. 예를 들어 visual clustering을 할 때 시간대 별로 따로 clustering을 시도한 다음 tag data를 통해 합친다거나, cluster내의 user id 종류가 적다면 landmark가 아닐 가능성이 높다는 점을 활용할 수 있었을 것이다.

<Reference>

- 1) Lowe, G. "Sift-the scale invariant feature transform." Int. J 2.91-110 (2004): 2.
- 2) S. Takumi and S. Miyamoto, "Top-down vs bottom-up methods of linkage for asymmetric agglomerative hierarchical clustering," 2012 IEEE International Conference on Granular Computing, 2012, pp. 459-464,

Meanshift

https://en.wikipedia.org/wiki/Mean_shift

silhouette score

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html

sift example code

https://docs.opencv.org/4.x/dc/dc3/tutorial_py_matcher.html

Word2Vec

<https://radimrehurek.com/gensim/models/word2vec.html>

<https://wikidocs.net/50739>