Technical University of Berlin

Fraunhofer Institute for Open
Communication Systems FOKUS

Faculty IV

Projekt Vertiefung / Vernetztes und
Automatisiertes Fahren
Winter term 2022/2023

Fast Analysis of Image Collection

Team members:

Yiwei Diao (460898)

Yilei Chen (451619)

Supervised by:

Jens Pontow

March 1, 2023

Contents

1 Sperrvermerk

Der vorgelegte Praktikumsbericht mit dem Titel [Fast Analysis of Image Collection] beinhaltet vertrauliche Informationen und Daten des Instituts **Fraunhofer Institut für offene Kommunikatioinssysteme, FOKUS sowie des DCAITI der TU Berlin**.

Dieser Praktikumsbericht darf nur vom Gutachter sowie berechtigten Mitgliedern des Prüfungsausschusses eingesehen werden. Eine Vervielfältigung und Veröffentlichung des Praktikumsberichts ist auch auszugsweise nicht erlaubt.

Dritten darf diese Arbeit nur mit der ausdrücklichen Genehmigung des Verfassers und Instituts zugänglich gemacht werden.

2 Project Topic

2.1 Motivation

As we all know, image annotation plays a very important role in computer vision. Efficient and accurate image annotation can help us train more powerful neural networks in deep learning. But at present, image annotation is basically done manually which is daunting and time-consuming. Therefore we should label the image in a more efficient way. So our Motivation is to save time and manual effort during image annotation and achieve a better fitting model.

2.2 Scope of your project

The main task of our project is to figure out which images should be labeled first, and which images contribute more to training our model. Kind of like evaluating the images dataset. And at the end of the project, we are supposed to get the score for each image and its ranking, the images with higher scores should be labeled with priority.

3 Project Progression

In the beginning, we use 3 existing tools for image analysis. Which are fastdup, imagededup, and fiftyone. Each of them has its own way of analyzing the images without any label.

3.1 Fiftyone

With fiftyone, we could get the Uniqueness. A score measuring the distribution of each sample in the dataset to find near-duplicate images(Figure 1). And it has the visualization capability that allows it to generate a Histogram of the uniqueness field of the samples and objects in the dataset(Figure 2). It could also be used to query datasets by location(Figure 3).

Figure 1: Uniqueness

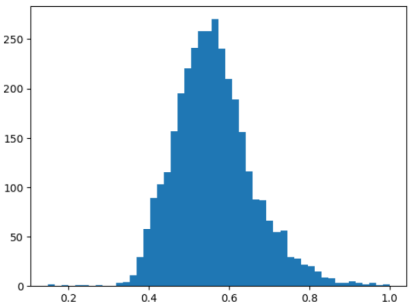

Figure 2: Histogram of the uniqueness field

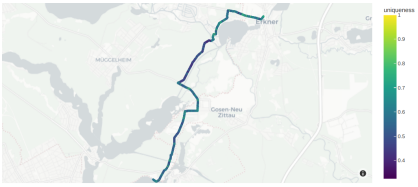

Figure 3: Map

3.2 Fastdup

We also tried fastdup. The next few figures show the experiment results for Image analysis by using fastdup. There are some outliers(Fig.4), which look strange or uncommon. And there is brightness(Fig.5), showing some darkest and brightest images within the dataset. And the blurriness of the images is displayed in Figure 6. The most blurry and sharpest images are shown here.

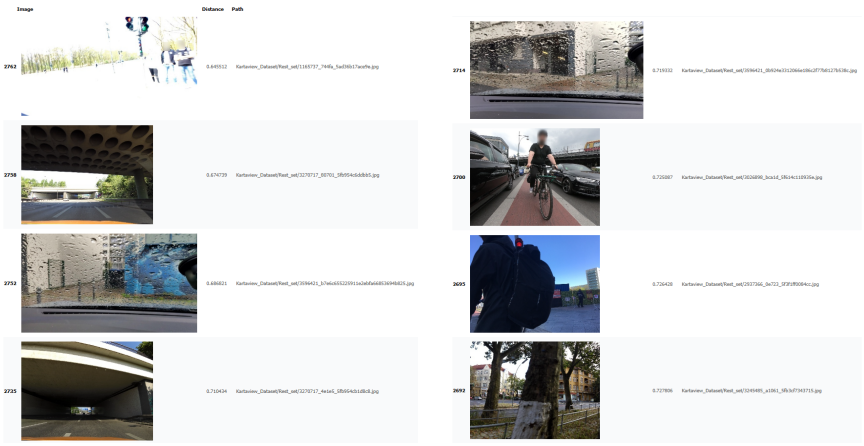

Figure 4: Outliers

As fastdup has more functions for image analysis and imagededup is similar to fiftyone. So after comparison, we decide to use fastdup mainly and combined it with fiftyone.

3.3 Kmeans clustering and punished values

Then we considered another situation, where a small set of data are already labeled. At this point, we came up with the idea of using KMeans clustering. We cluster the images first and for each cluster if there are labeled data within the cluster. Then, for all data within the cluster. We plan to give them a punished value, meaning they are not worthy to be labeled first.

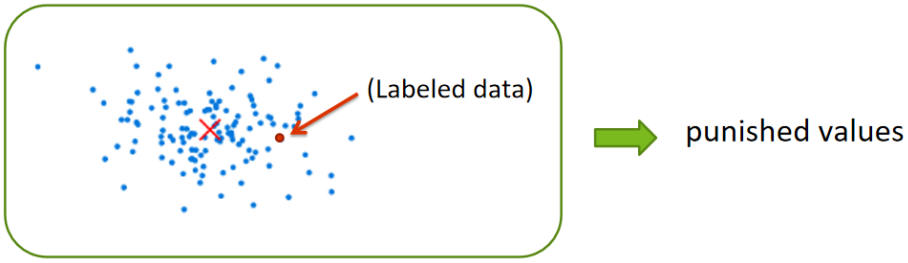

Figure 7: Kmean

3.4 Kmeans clustering and elbow Function

And as we all know, it is important for the KMeans algorithm to choose the k value. For doing that, we use the elbow function at first and got the elbow curve like this. (Fig.8) It is ambiguous where the elbow is, so we decide to use

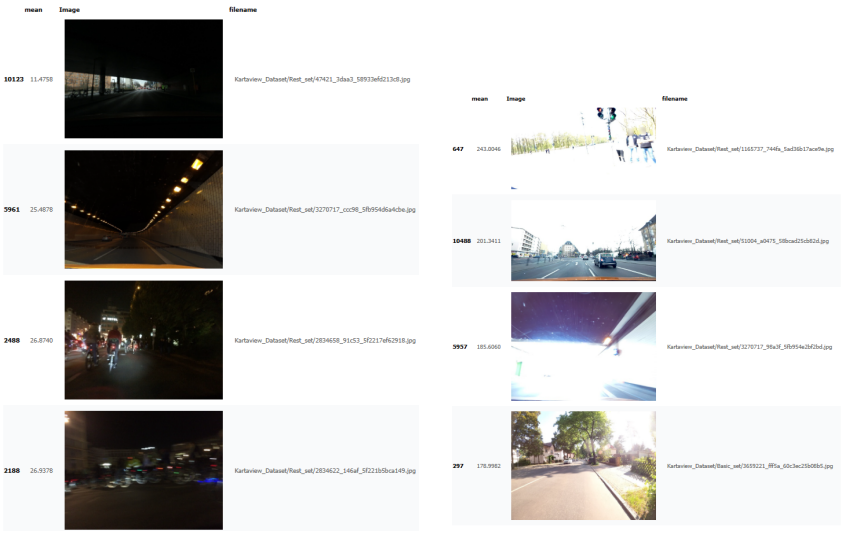

Figure 5: Brightness

python's integrated elbow function and silhouette function module for a more precise result. Figure 9 is the result of KMeans clustering with k equal to 26. And at the next step, after we get the indicators, We need to decide which of them should be used for calculating the score and how much weight should be set.

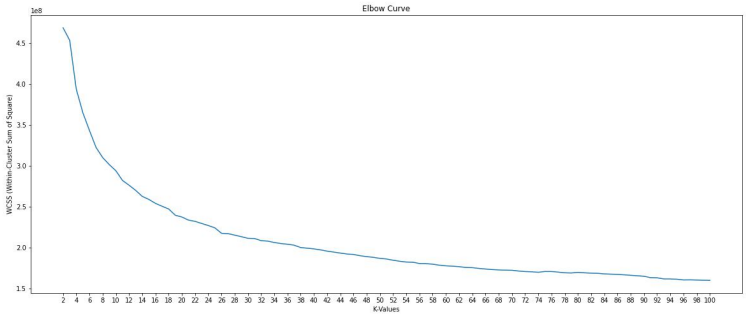

Figure 8: Elbow Curve

Figure 6: Blurriness

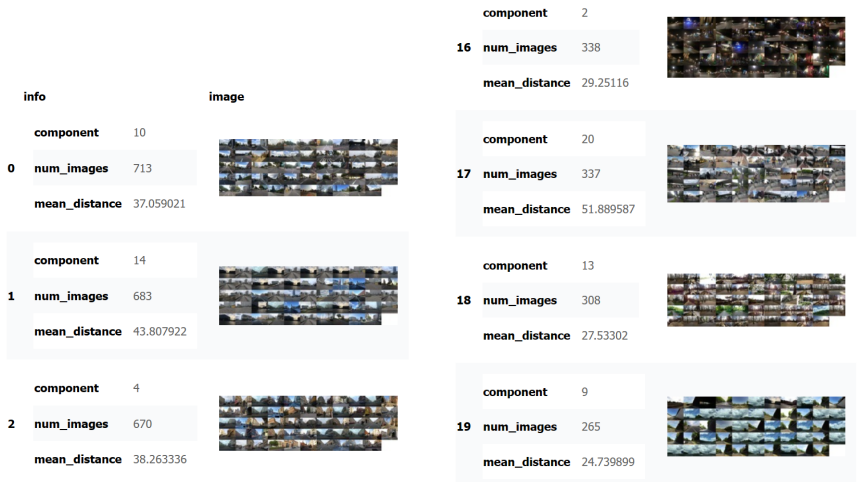

Figure 9: Result of Clustering

4 Current Status

4.1 Download Images

After exploring different tools on dataset 'fahrradtour2022', we find that this dataset's size is a bit small and therefore insufficient for further process. So under the supervisor's suggestion, we find much better images as our new dataset from Kartaview, which is a website that collects and shares street-level images. Not only can we directly view images on the website, but we also can use a plugin to display these along with other layers, which gives us a better illustration.

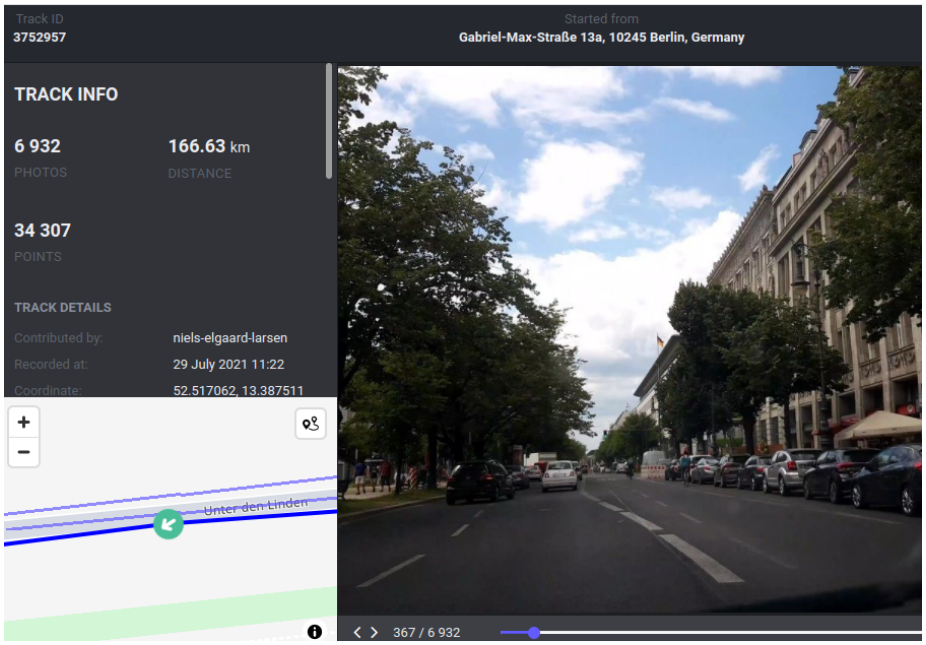

Figure 10: Kartaview

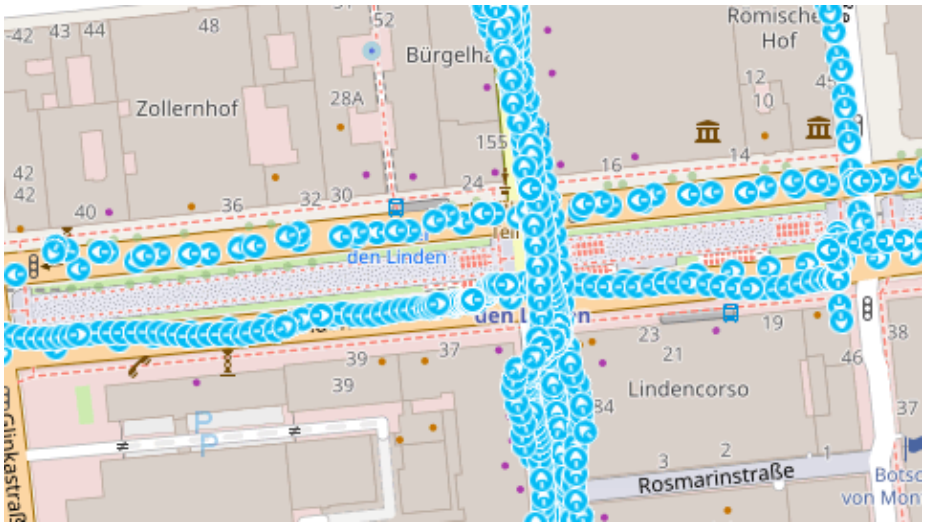

Figure 11: Java OpenStreetMap Editor

Since we can not download images massively from Kartaview, we write some python files to help us. Here's our idea:

1. Firstly, `osc_seq_from_geo.py` file is created. After defining the basic URL, a function '`sequence_of_photo`' is used to generate the related sequences based on the image's latitude and longitude coordinates. Furthermore, we allocate one coordinate for labeled images and another three coordinates for unlabeled images because one of this project's premises is that 5% images have already been labeled and should be different from unlabeled images. Then we can get the corresponding sequence ids based on their coordinates.

2. Secondly, in `osc_get_image_address.py` file two functions are generated: the first one is used to generate the sequence's endpoint with sequence id, page number, and items per page; the second one can use endpoints to generate all images' URLs within. Then, after importing results from step one, we get labeled and unlabeled image URLs respectively.

3. Lastly, we download images based on their URLs and store them in different folders in `osc_download_image.py` file.

After implementing the commands above, we get 11084 images in total. Among them, 598 images are assumed to have been labeled and stored in folder `Basic_Set`. The rest 10486 images are unlabeled, stored in folder `Rest_Set`, and need to be ranked.

4.2 Separate Images

Since the new dataset's size is quite large and it's impossible to visualize it all at one time, we look up all the functions Fastdup provides and think that Kmeans would be a good method to help us separate images. But first of all, we need to determine which number of clusters is suitable for our dataset. Technically, there are several categories of methods for making this decision.

For example, the elbow method looks at the percentage of explained variance as a function of the number of clusters: one should choose a number of clusters so that adding another cluster doesn't give much better modeling of the data. So in the graph, we need to find an obvious angle and pick the number of clusters at this point as the "elbow criterion". However, in most datasets, this "elbow" is ambiguous, making this method subjective and unreliable.[1]

The average silhouette of the data is another useful criterion for assessing the natural number of clusters. In order to calculate the silhouette score for each image, the following distances need to be found out for each image belonging to all the clusters:

- Mean distance between the observed image and all other images in the same cluster. The distance can also be called mean intra-cluster distance. The mean distance is denoted by a.
- Mean distance between the observed image and all other images of the next nearest cluster. This distance can also be called as mean nearest-cluster distance. The mean distance is denoted by b.

Silhouette score, S, for each sample is calculated using the following formula:

$$S = \frac{(b - a)}{max(a, b)} \tag{1}$$

The value of the silhouette score varies from -1 to 1. If the score is 1, the cluster is denser and well-separated than other clusters. A value near 0 represents overlapping clusters with samples very close to the decision boundary of the neighboring clusters. A negative score [-1, 0] indicates that the sample might have got assigned to the wrong clusters.[2]
Since python already has a module that implements KMeans and some corresponding methods for picking the appropriate number of clusters, we can simply call them after we import feature vectors. Thankfully, fastdup can easily compute a dataset's feature using the function `load_binary_feature()`, which generates 576 vectors for each image by default. We first realize the elbow method by setting k between 2 and 100.

```
# Load binary feature
karta_file_list, karta_mat_features =
    fastdup.load_binary_feature("./Karta_out_2/atrain_features.dat")
```

Listing 1: Function load_binary_feature()

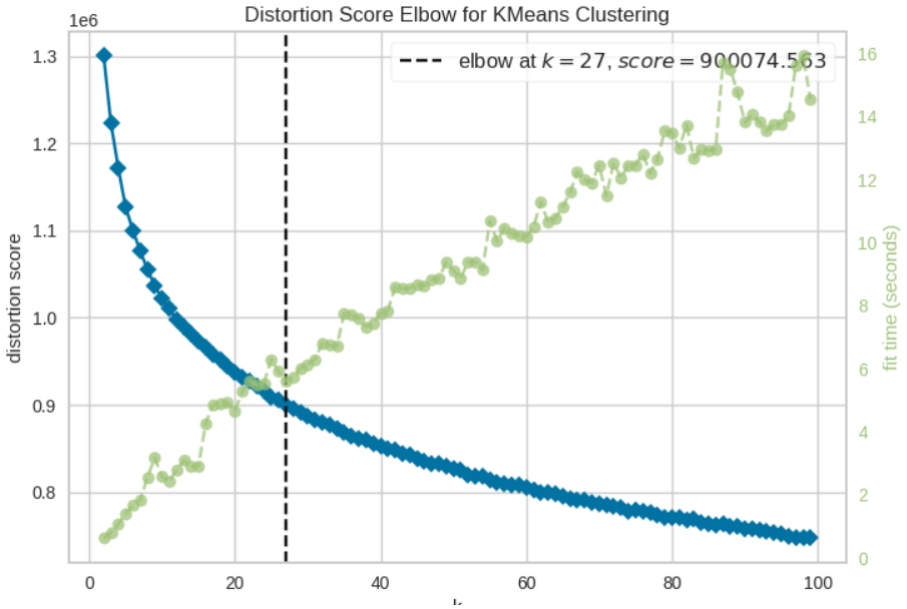

Figure 12: Elbow function with k = (2, 100)

From the figure 12 above we can see that there is no obvious "elbow criterion", although it suggests that the elbow is at $k = 27$. So we try to expand the range of k to 200 to see if the elbow is larger than 100. The result is shown below, which says that the elbow is at $k = 45$, while it is still not an obvious point. But if we draw lines along this curve's tendency, we can get that about at $k = 12$ and $k = 27$ this curve's gradient changes drastically. Compared to $k = 12$, $k = 27$ has more clusters, which means the clusters' size is smaller, making it easier to be implemented and visualized. So we consider $k = 27$ might be a plausible number of clusters.

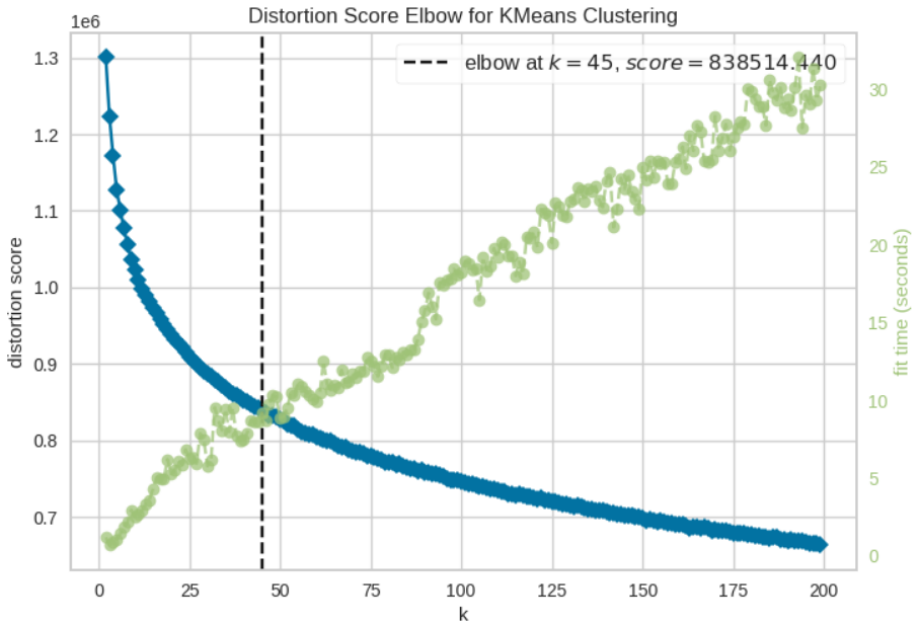

Figure 13: Elbow function with k = (2, 200)

Now that we choose 27 as the "elbow criterion", we need to testify to its rationality with the help of the silhouette method. We create SilhouetterVisualizer instances with KMeans instances for $k = [25, 30]$ and the result is shown in Figure 14.

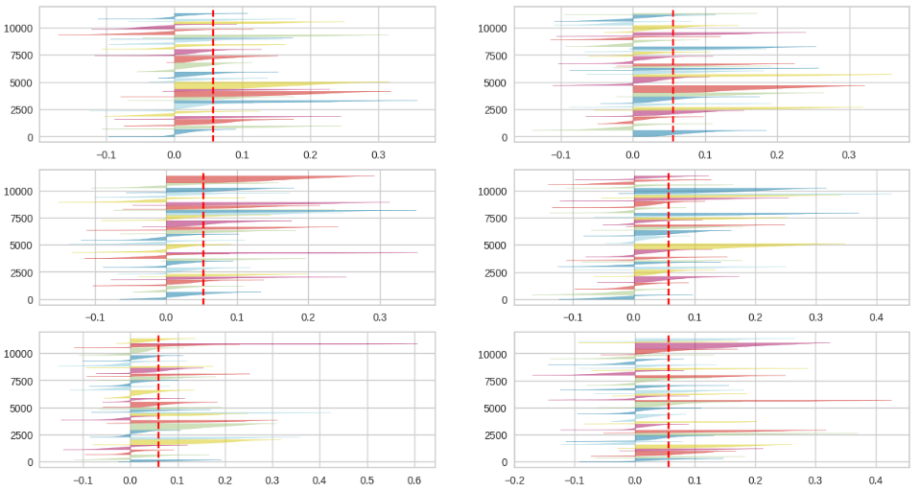

Figure 14: Silhouette score with k = (25, 30)

As we can see when k is between 25 and 30, the average score is about 0.05, which means that there are some overlapping clusters. What's worse, some clusters' scores are negative, indicating that some samples might belong to the other clusters. These scores also reflect that this dataset is not appropriate to be separated by KMeans. Despite all this, we still use KMeans to cluster our dataset because we need to split the dataset into smaller units for better visualization. After discreetly comparing these outputs, we think that 26 might be a more plausible number because its negative scores are a little smaller than scores at $k = 25, 27, 28, 30$ and its positive scores are more average than the others. After importing 26 in Fastdup's KMeans function, it returns the cluster centroid and each sample's path, cluster id, and distance to the centroid.

```
# run KMeans by fastdup
fastdup.run_kmeans(input_dir = "./Kartaview_Dataset", work_dir =
    'Karta_out', num_clusters = 26, nearest_neighbors_k = 1)
```

Listing 2: Function run_kmeans()

4.3 Define Parameters

4.3.1 Uniqueness

According to this project's requirements, we need to rank images mainly based on similarity because if we put similar images in the training phase of DNNs, it will not improve the model's performance as well as some different images. In other words, unique images should be ranked higher than similar images.

In the first half of this project, we explored different tools for the image process. Among them, Fiftyone has a simple function to compute each image's uniqueness score in the range of 0 and 1, which only uses pixel data and can therefore process labeled or unlabeled samples. After creating a data frame for storing each sample's path and cluster id, we split this data frame into multiple data frames based on cluster id. And then in each cluster, we compute each sample's uniqueness value.

```
# uniqueness function
def compute_uniqueness(image_address):
    # load image into Fiftyone based on DataFrameDictNew value
    dataset = fo.Dataset.from_images(image_address)
    # compute uniqueness
    fob.compute_uniqueness(dataset)
    # save filepath and uniqueness score as array 'uniq'
    uniq = np.empty((0,2))
    for sample in dataset:
        temp = np.array([[sample["filepath"], sample["uniqueness"]]])
        uniq = np.append(uniq, temp, axis=0)
    # convert array 'unique' to dataframe 'df_temp'
    df_temp = pd.DataFrame(uniq, columns=['filepath', 'uniqueness'])
```

```
    # rank based on uniqueness score
    df_temp.sort_values("uniqueness", ascending = False, inplace =
        True)
    return df_temp
```

Listing 3: Function compute_uniqueness()

4.3.2 Uniqueness_punishment

Although this uniqueness method is quite simple and straightforward, it gives similar images with very close scores, which means that we can't distinguish similar images solely relying on this score. We need to select one image from a similar group as the representation and adjust its rank higher than the other ones. In order to realize this goal, we use function `create_similarity_gallery()` from Fastdup. This function generates a similarity data frame, which for each image filename returns a list of top similar images. So we call this function in each cluster and get the images' similarity information respectively. Based on this similarity function, we can get similar groups in each cluster. Then we introduce a parameter: *uniquenesspunishmentscore* and set only one sample's score as 1 in each similar group and the rest as 0.5.

```
# generate uniqueness punishment score based on cluster id
def uniqueness_punishment_score(df_simi, results_dir, cluster_id):
    df_uniq = pd.DataFrame(columns=['filepath', 'uniqueness
        punishment'])

    for i in range(len(df_simi)):
        # check if item from 'from' column already exists in 'df_temp'
        # if yes, then pass; if not, then save it and its punishment
            value to 'df_temp'
        if df_simi['from'][i] in df_uniq['filepath'].values:
            pass
        else:
            row_from = [df_simi['from'][i], 1.0]
            df_uniq.loc[len(df_uniq)] = row_from

        for j in range(len(df_simi['to'][i])):
            # check if item from 'to' column already exists in
                'df_temp'
            # if yes, set the latest punishment value as 0.5; if not,
                pass
            if df_simi['to'][i][j] in df_uniq['filepath'].values:
                df_uniq.at[len(df_uniq)-1, 'uniqueness punishment'] =
                    0.5
            else:
                pass

        for j in range(len(df_simi['to'][i])):
```

```
            # check if item from 'to' column already exists in
                'df_temp'
            # if yes, pass; if not, then save it and its punishment
                value to 'df_temp'
            if df_simi['to'][i][j] in df_uniq['filepath'].values:
                pass
            else:
                row_to = [df_simi['to'][i][j], 0.5]
                df_uniq.loc[len(df_uniq)] = row_to

    # export 'df_uniq' as
        'cluster_id_uniqueness_punishment_score.csv'
    df_uniq.to_csv(results_dir +
        '/cluster_'+str(cluster_id)+'_uniqueness_punishment_score.csv')
```

Listing 4: Function uniqueness_punishment_score()

Figure 15: Uniqueness_punishment(only one score as 1, the rest as 0.5)

4.3.3 Outlier_distance

In fastdup, there is also another interesting function: `create_outlier_gallery()`. This function can detect outliers based on similar distances and return the corresponding outliers' distances. And this situation could happen: an image's uniqueness score is high, which is good, but it could also be counted as an outlier, which is bad. So in order to conquer this disadvantage of the uniqueness function, we use this function in each cluster to find outliers and integrate *outlierdistance* into the final rank. For those images that are not outliers, their outlier distance is 0.

```
# generate outliers based on cluster id
def fastdup_outlier_gallery(results_dir, cluster_id):
```

```
outlier_gallery_save_path = os.path.join(results_dir,
    'outlier-gallery')
if not os.path.exists(outlier_gallery_save_path):
    os.makedirs(outlier_gallery_save_path)

fastdup.create_outliers_gallery(results_dir,
    save_path=outlier_gallery_save_path)

df_outl = pd.DataFrame(columns=['filepath', 'outlier distance'])

df_outl_temp = pd.read_csv(results_dir+"/outliers.csv")

for i in range(len(df_outl_temp)):
    # check if item from 'from' column already exists in 'df_outl'
    # if yes, then pass; if not, then save it and its outlier
        distance to 'df_outl'
    if df_outl_temp['from'][i] in df_outl['filepath'].values:
        pass
    else:
        row_from = [df_outl_temp['from'][i],
            df_outl_temp['distance'][i]]
        df_outl.loc[len(df_outl)] = row_from

# export 'df_outl' as 'cluster_id_outlier_distance.csv'
df_outl.to_csv(outlier_gallery_save_path +
    '/cluster_'+str(cluster_id)+'_outlier_distance.csv')
```

Listing 5: Function create_outliers_gallery()

Figure 16: Outlier example(Uniqueness score: 0.74, Outlier distance: 0.73)

4.3.4 Similar_distance

Because the KMeans method is not very accurate, we think it's better to find out which unlabeled images are similar to labeled images and give them a penalty. Here, we define a new parameter: *similardistance*. Then we read similar image pairs by Fastdup in each cluster, find only unlabeled images that are similar to labeled images, read their distances as similar distances, and set the rest images' similar distances as 0.

```
df_rest_punish = pd.DataFrame(columns=['Basic_set', 'Rest_set',
    'distance'])

for i in range(len(df_all_similarity.index)):
    if df_all_similarity['from'][i].split('/')[-2] == "Basic_set"
        and df_all_similarity['to'][i].split('/')[-2] != "Basic_set":
        new_row = [df_all_similarity['from'][i],
            df_all_similarity['to'][i],
            df_all_similarity['distance'][i]]
        df_rest_punish.loc[len(df_rest_punish)] = new_row
    elif df_all_similarity['from'][i].split('/')[-2] != "Basic_set"
        and df_all_similarity['to'][i].split('/')[-2] == "Basic_set":
        new_row = [df_all_similarity['to'][i],
```

```
            df_all_similarity['from'][i],
            df_all_similarity['distance'][i]]
        df_rest_punish.loc[len(df_rest_punish)] = new_row
    else:
        pass
```

Listing 6: Find images based on similarity.csv


Figure 17: Similar pair example(Similar distance: 0.92)

4.3.5 Blur and Brightness

Besides similarity, fastdup also provides `create_stats_gallery()` to compute images' information, such as blur, mean, size, etc. So we also pick blur and brightness as two other supplementary parameters. We first call this function for all images and convert the original value proportionally into the range between 0 and 1. The bigger the blur value is, the more unsurpassed the image. And the bigger the brightness value is, the brighter the image.

4.4 Rank

4.4.1 Score Formula

After we get the parameters, we then use the formula below to calculate the image's score.

$$\begin{aligned} Score = (&uniqueness * uniqueness\_punishment \\ &+ outlier\_weight * outlier\_distance \\ &+ similar\_weight * similar\_distance \\ &+ blur\_weight * blur \\ &+ brightness\_weight * brightness) \end{aligned}$$

And here are the weights we picked:

- `outlier_weight`: -0.15
- `similar_weight`: -1

- `blur_weight:` 0.2
- `brightness_weight:` 0.05

Since labeled and unlabeled similar pairs are something we don't want at all, we set *similar_weight* much smaller than *outlier_weight*. And an image's blurriness can be more important than its brightness, so *blur_weight* is bigger than *brightness_weight*.

4.4.2 Cluster Priority

Now that all parameters and weights are settled, we need to rank all images based on the score formula. However, we think a simple rank without considering the image's cluster id is inappropriate. So we first rank clusters based on labeled images within. We divide clusters into 4 groups: the first one has no labeled images at all, the second one has a few labeled images(each cluster's labeled images' number is smaller than 10 and percentage smaller than 3%), the third group has some labeled images(number<50 and percent<30%), and the last group has a lot of labeled images(number>50 ore percent>30%). Here's our result:

Cluster Rank	Cluster Id
1	2, 0, 15, 25, 14, 5, 4
2	3, 7, 17, 21, 20, 23, 24
3	18, 16, 8, 1, 22, 11
4	6, 13, 19, 9, 12

Table 1: Cluster Priority

4.4.3 Final Result

In the last step, we first delete labeled images' parameters and calculate the unlabeled images' scores. We then separate these scores into 3 regions: good(bigger than 0.8), medium(between 0.8 and 0.2), and bad(smaller than 0.2). Then we can see that good and bad image numbers are 580 and 686. The medium region has 9,220 images. In each region, we first rank scores based on cluster priority. Then in each cluster rank group, we rank scores based on their values. At last, we simply combine each region's rank into one whole rank and save it as $all\_score\_after\_rank.csv$ file.

5 Outlook

Although we get a total rank of all images, our approach still has some problems and can be improved. For example,

1. Image download should be addressed: The current method is to use the image's coordinates to access the corresponding sequence and then download. A better method would be combining coordinates and ranges to get all images in a certain region.

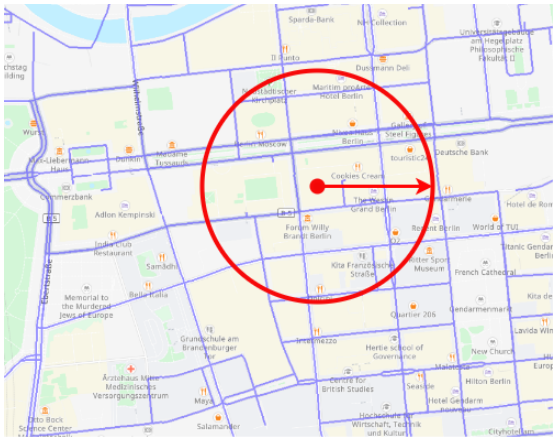

Figure 18: Image Selection

2. The silhouette function shows that KMeans might not be a good solution for this dataset's separation. It did cluster some similar images into one cluster but samples are close to the decision boundary of the neighboring clusters. So a better separation method should be explored.

3. Our parameters and weights are quite subjective since there are no standard criteria for this process. So when it comes to a different dataset, different parameters and weights need to be chosen.

References

[1] Wikipedia contributors. *Determining the number of clusters in a data set*. Feb. 2023. URL: https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set.

[2] Ajitesh Kumar. *KMeans Silhouette Score Explained with Python Example*. Sept. 2020. URL: https://vitalflux.com/kmeans-silhouette-score-explained-with-python-example/.