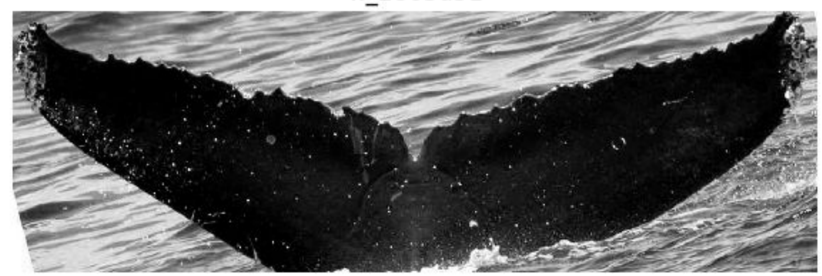
Humpback Whale Identification

by Korhan Polat for CMPE58Z
May 18, 2018

Problem definition: identify a whale by the picture of its fluke.

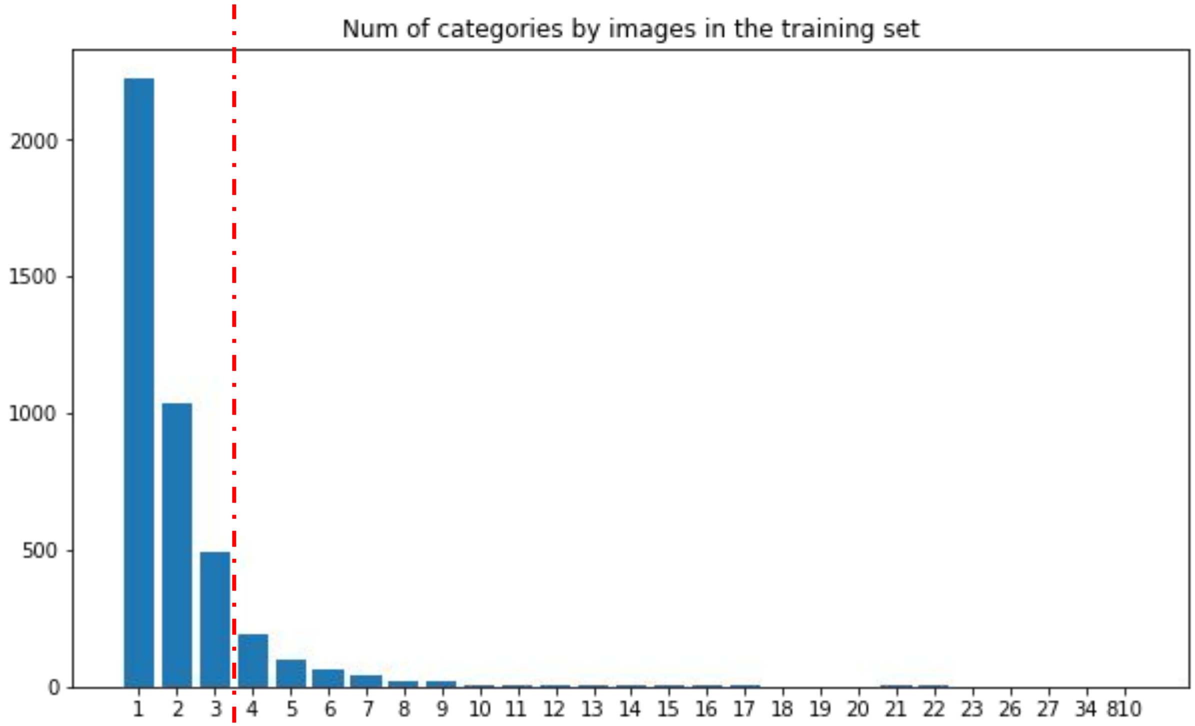
w_da2efe0

w_2863d51

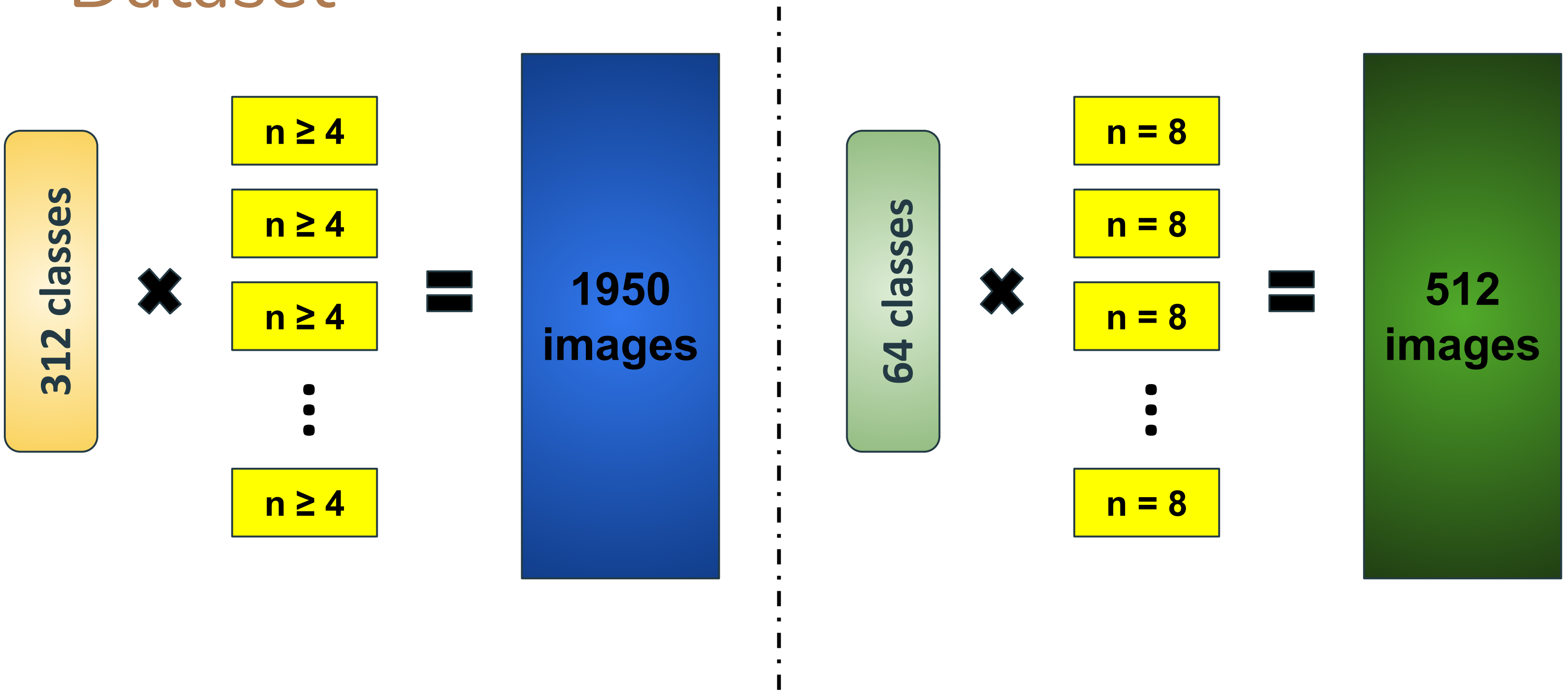


Dataset

- Dataset is provided by Kaggle.com
- Same side of the fluke appears for each whale class
- Originally 9850 images of 4.5k different whales
- Discarded images with bad enrollment and classes that has less than 4 images

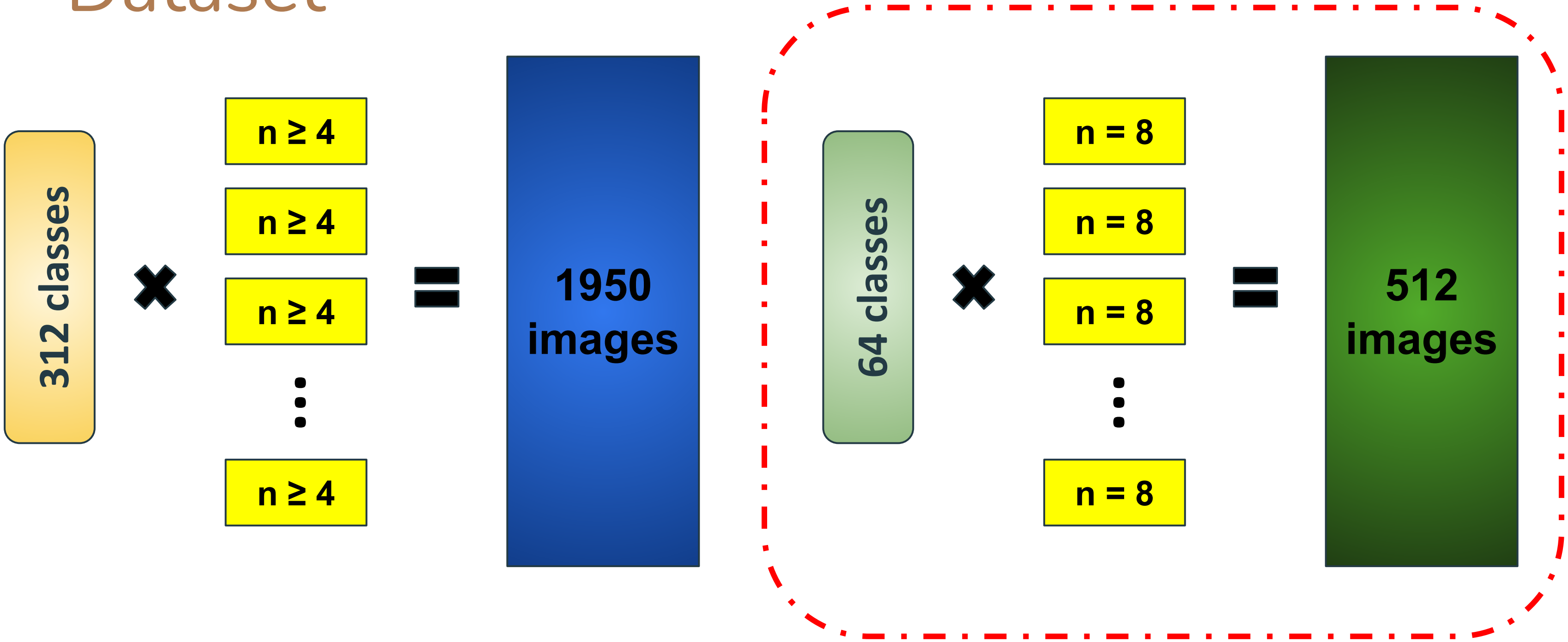


Dataset



- Set up two different experiments:
    1. **312** classes with **1950** images (imbalanced classes)
    2. **64** classes with **512** images (balanced classes)

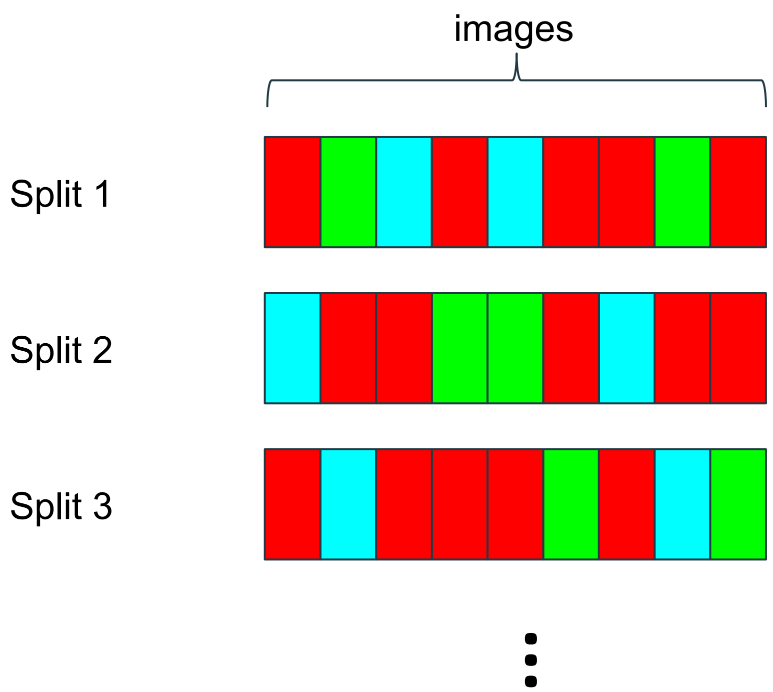
Dataset



- Tried the models on 2nd experiment set, because it is smaller and faster

Cross-validation

- Divided into 10 splits with
  - 50% training
  - 25% validation
  - 25% test
- Results are averaged over 10 splits



My approach

Preprocessing

1. Image enhancement
2. Manual cropping
3. Convert to grayscale
4. Gaussian smoothing

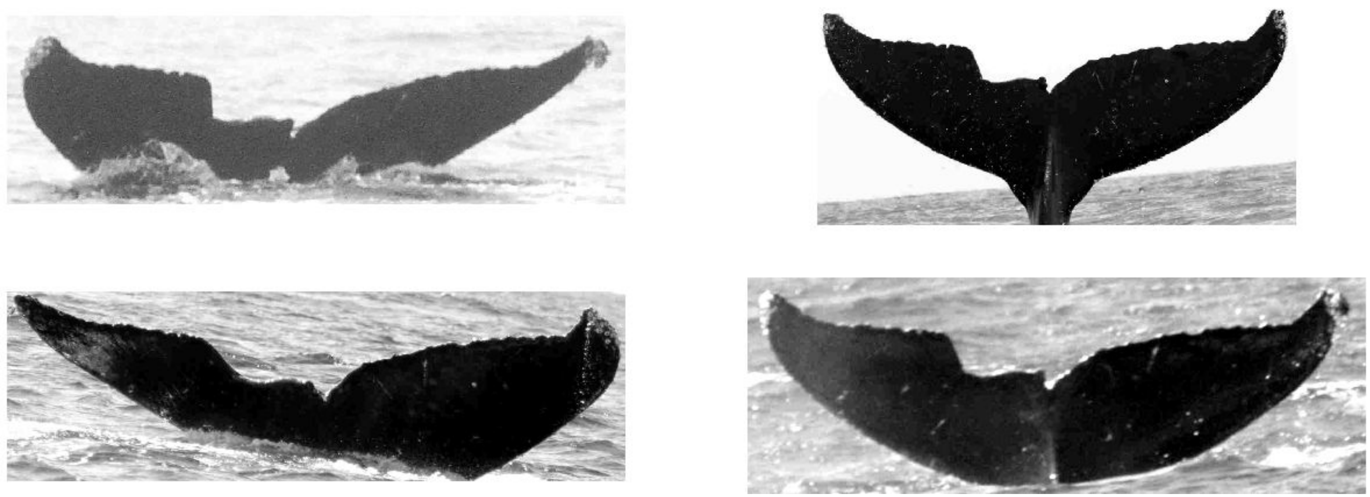
Feature extraction

5. Contour similarity
6. Eigen flukes
7. SIFT, ORB feature matching

Contour Similarity

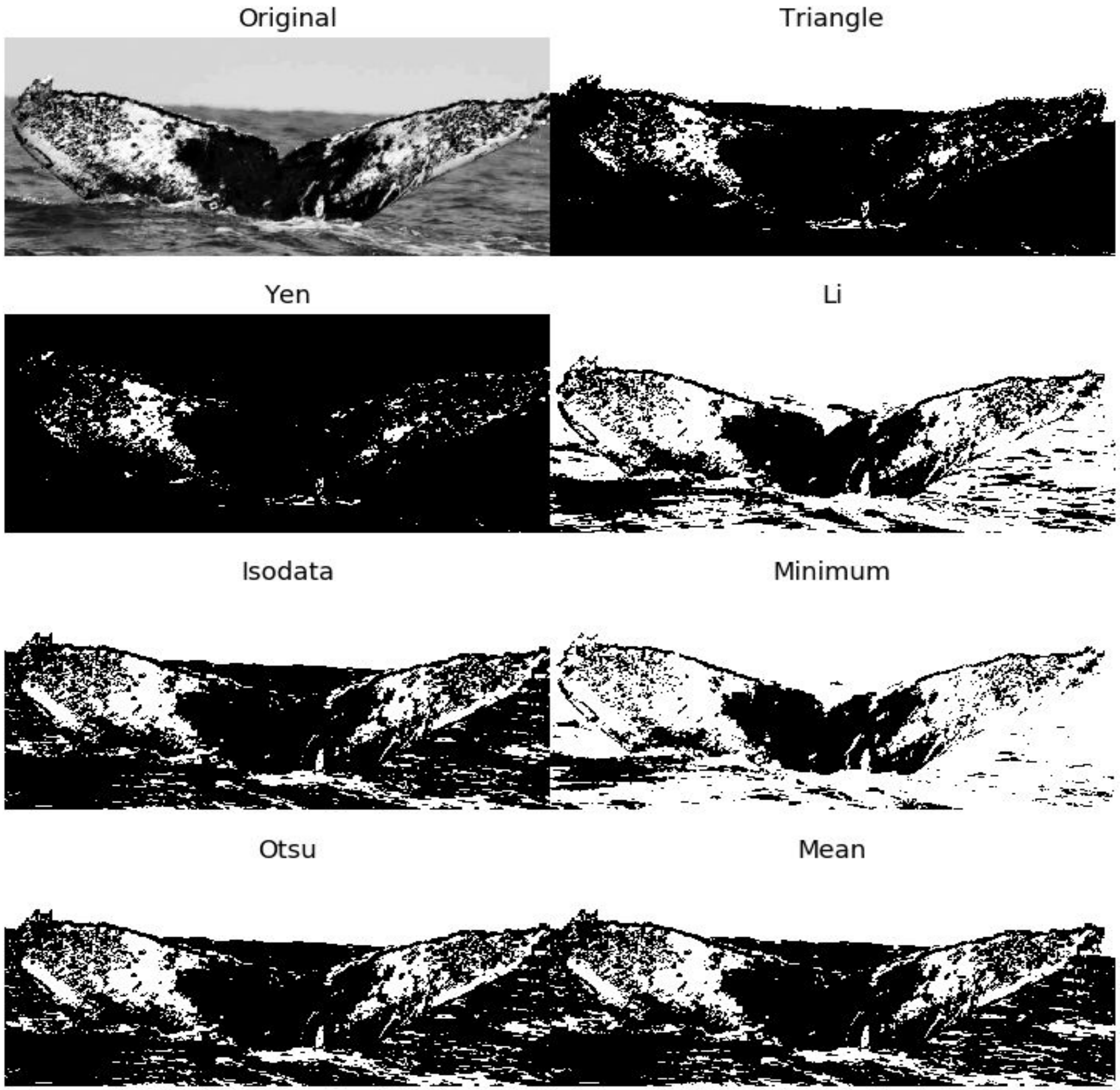
Contour Similarity

Observation: shape/contour of the fluke might be a good feature



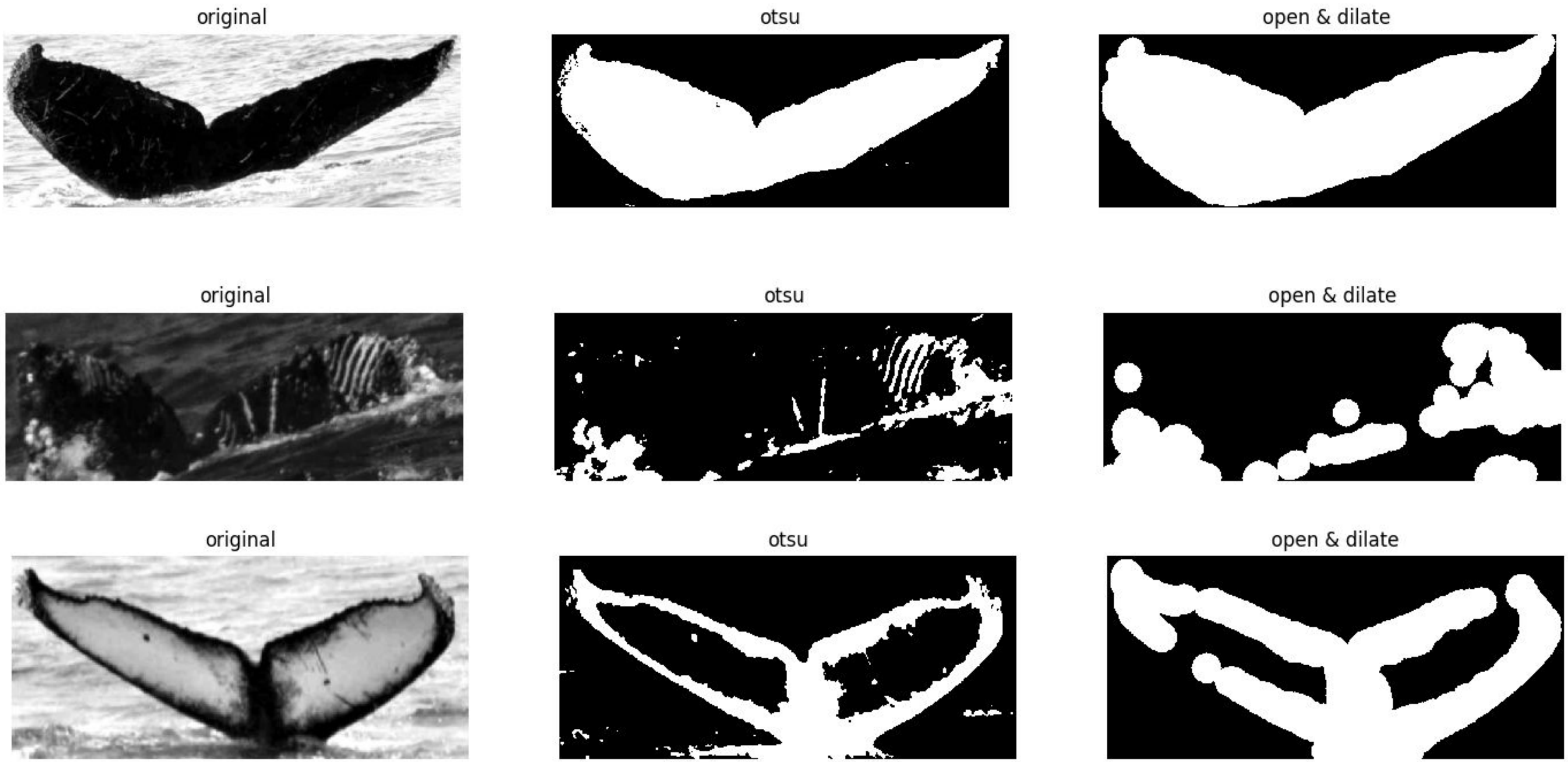
whale ID: ‘w_3b0894d’

Thresholding trials



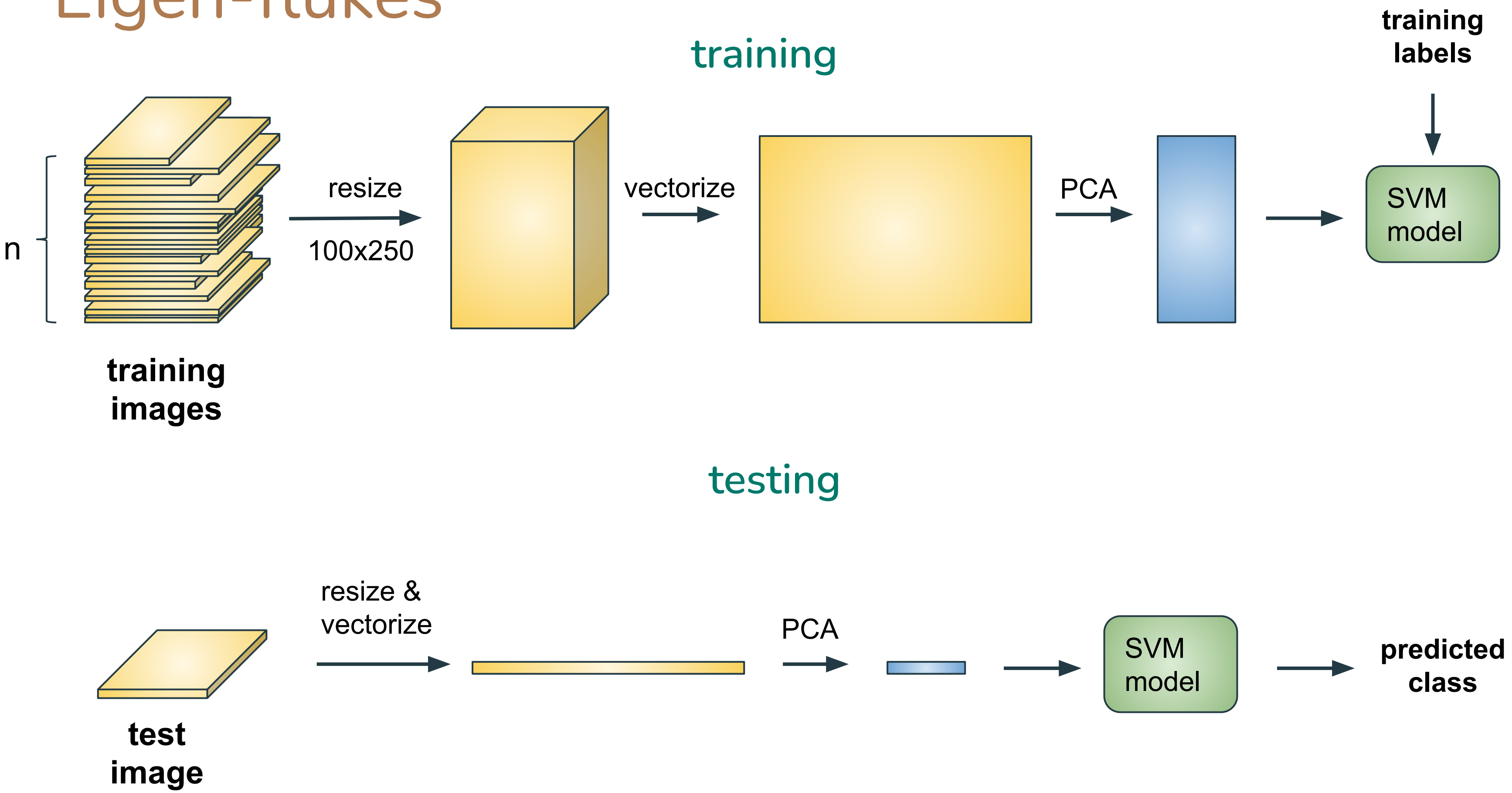
using *Scikit-Image* Python library

Thresholding failed...



Eigen flukes

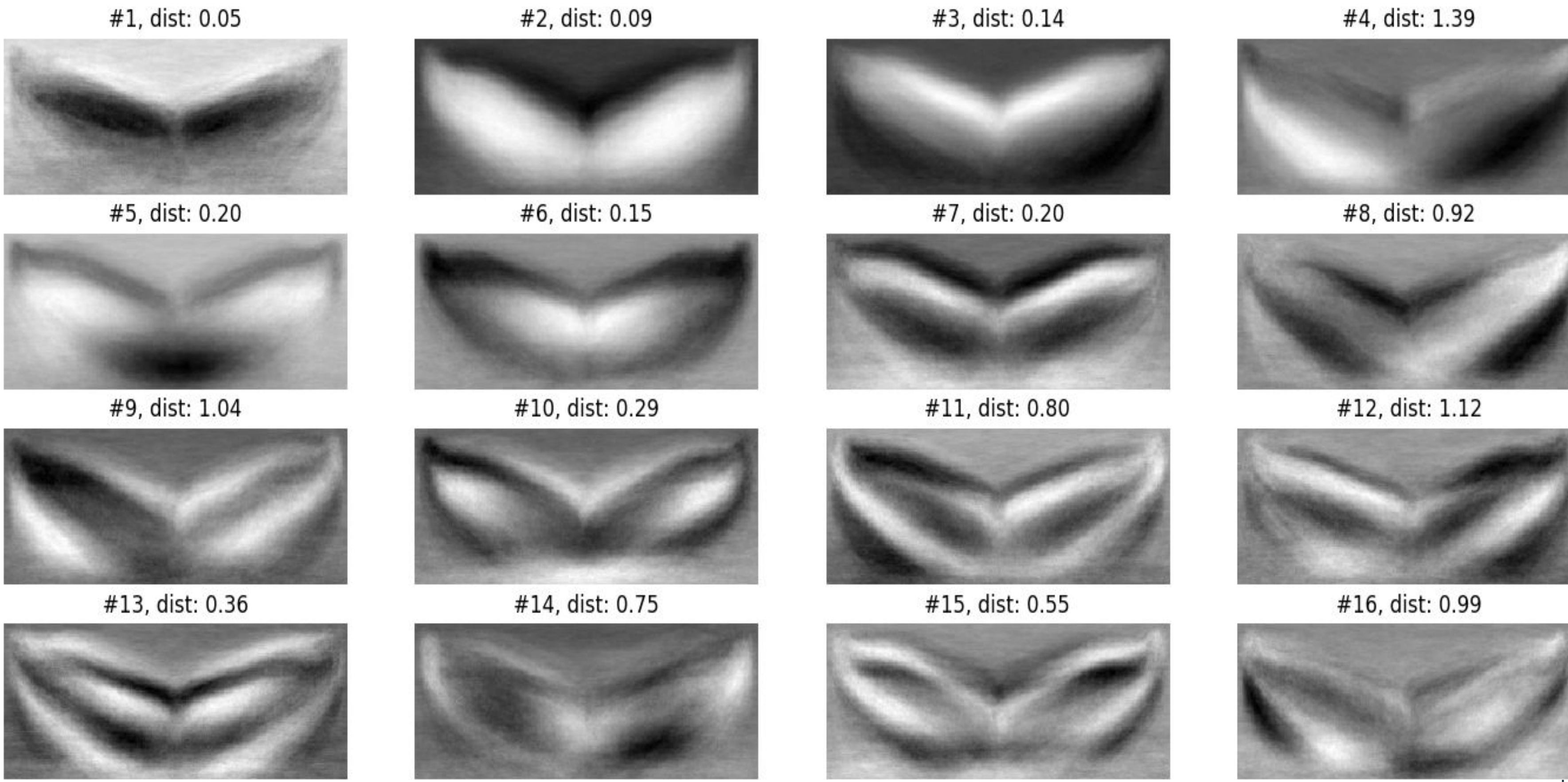
Eigen-flukes



** PCA and SVM is performed using Scikit-Learn Python library*

Eigen-flukes

- Defined a symmetry distance and eliminated asymmetric eigenvectors



Eigen-flukes

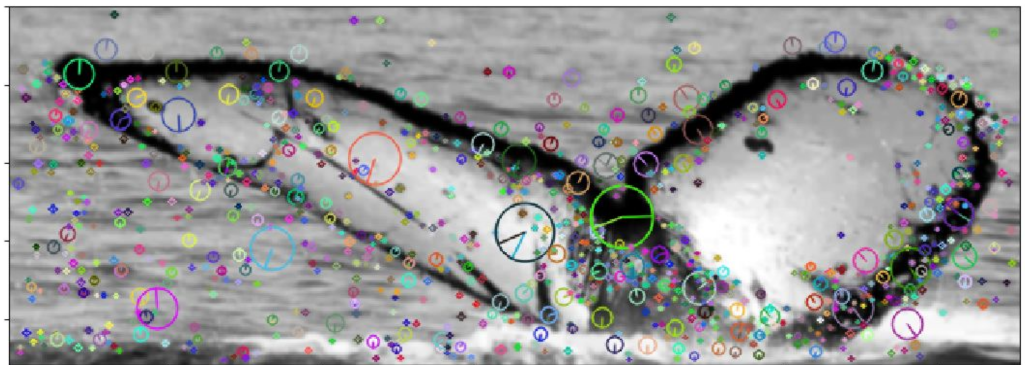
64 class, 256 training 128 test images
10-fold Cross Validation

Using Symmetry correction			Without Symmetry correction	
distance threshold	Average # components	Accuracy	# components	Accuracy
0.95	37.2	8.1%	37	9.1%
0.97	61.3	9.8%	61	11.6%
0.99	128.7	12.3%	129	12.5%
1.01	219.6	14.5%	220	13.7%
1.03	**245.1**	**15.0%**	245	14.1%
1.05	249.2	14.5%	249	14.5%
1.07	251	14.1%	251	14.4%
1.09	252.3	14.1%	252	14.9%
1.11	253	14.1%	253	14.8%
1.13	253.5	14.0%	256	14.8%
1.15	254	14.4%		

SIFT Features Matching

SIFT

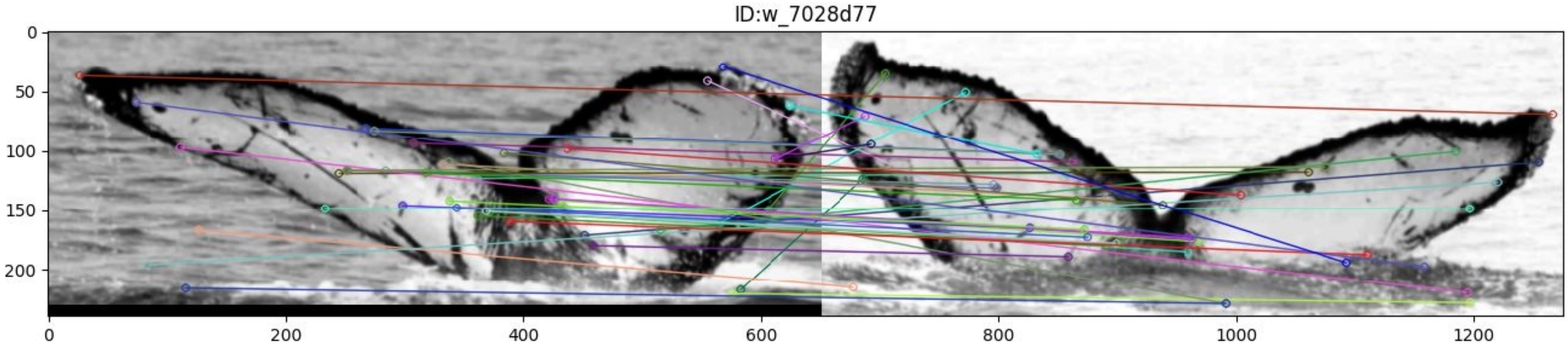
- Given an image, SIFT algorithm finds n interest points and outputs two variables for each of these interest points
    - Key points ⟶ (x , y , scale , angle)
    - Descriptor ⟶ a 128 dimension vector, describing the keypoint



**using Python OpenCV Contrib 3*

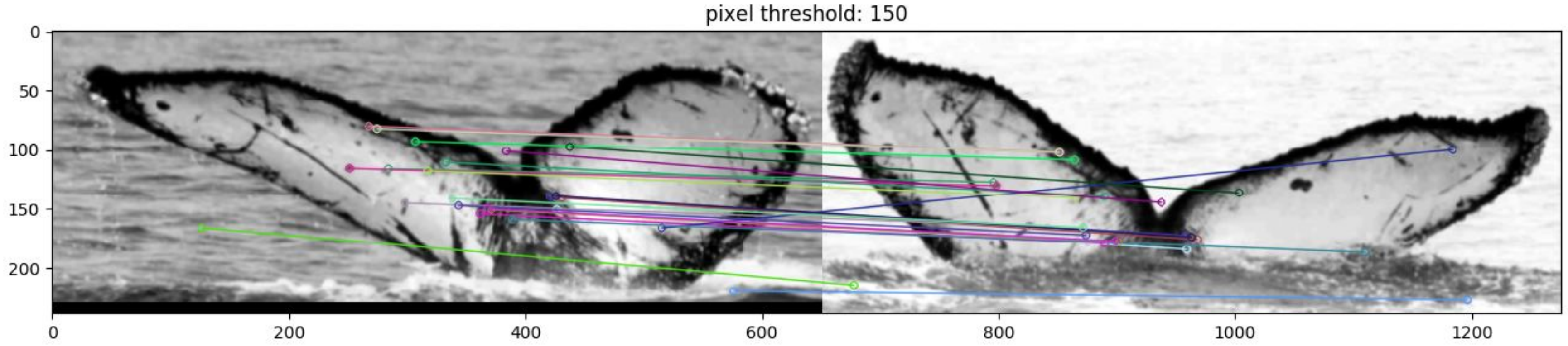
Feature matching

- OpenCV Python brute-force feature matcher:
  - Given a query image and training image descriptors, find descriptors that are close to each other (Euclidean dist)
- Matching descriptors are further eliminated with **ratio test**
  - Find 2 closest training descriptors to the query descriptor at hand
    - If *dist(m1) < dist(m2) x ratio* → store *m1* as a match



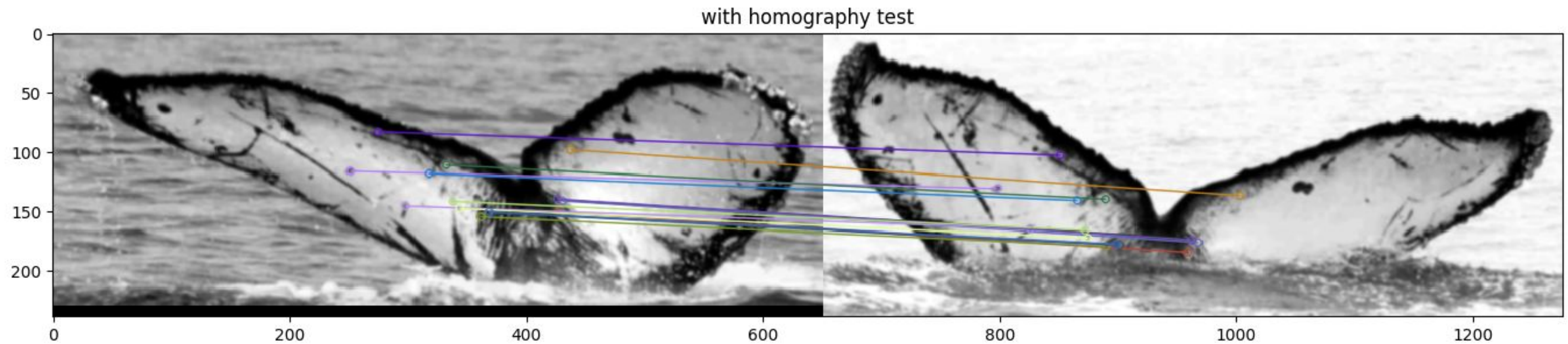
Matched keypoint distances

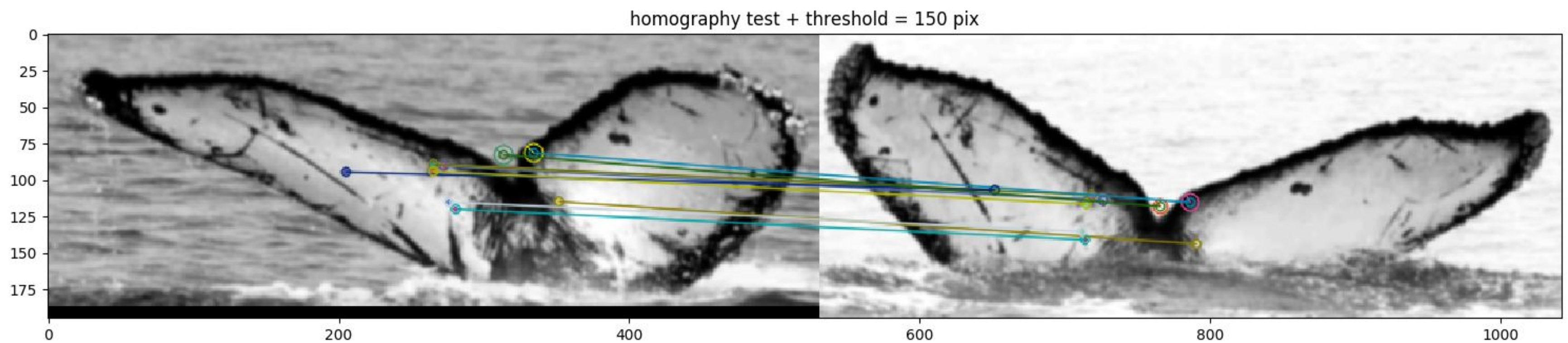
- Resized all images such that all have same number of pixels
- After finding matches with Brute-force matcher and ratio test, remaining matches are further eliminated according to pixel distance threshold



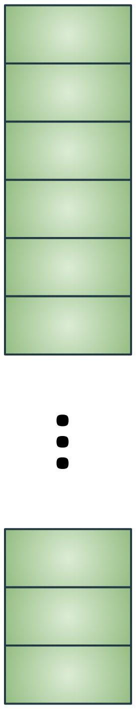
Homography test

Remaining matches are further eliminated with homography test, which takes into account spatial consistency of matched points



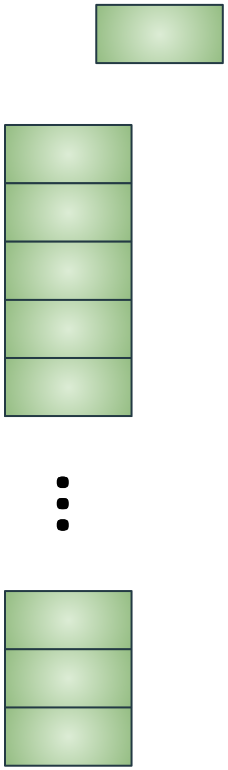


Training

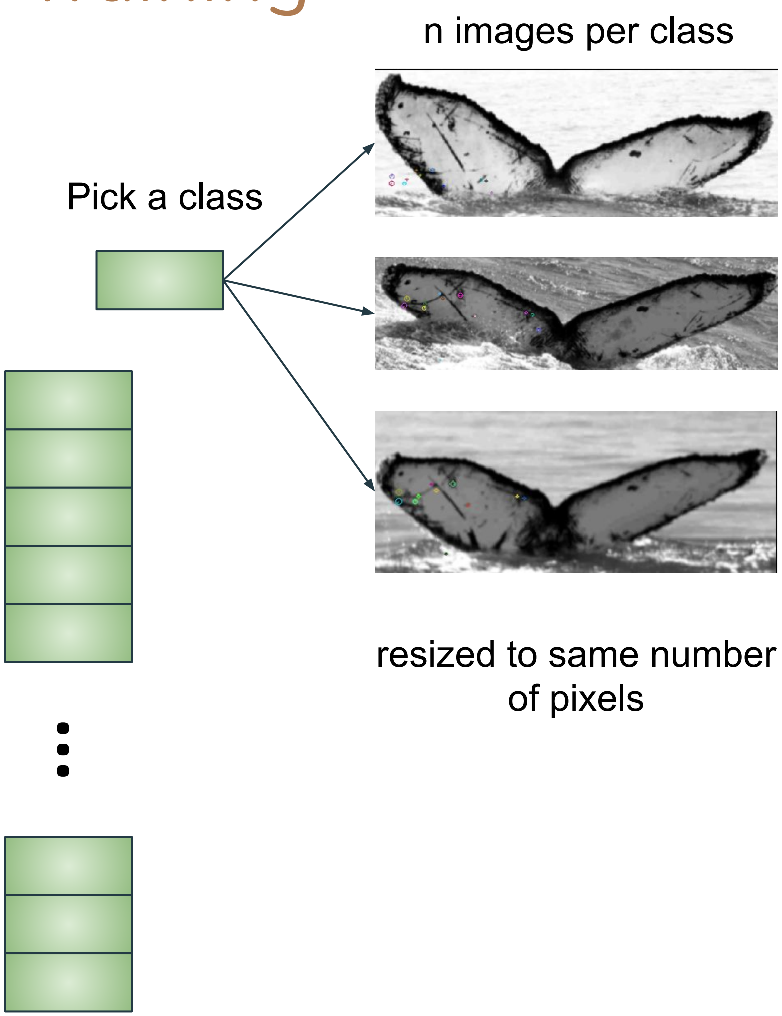


Training

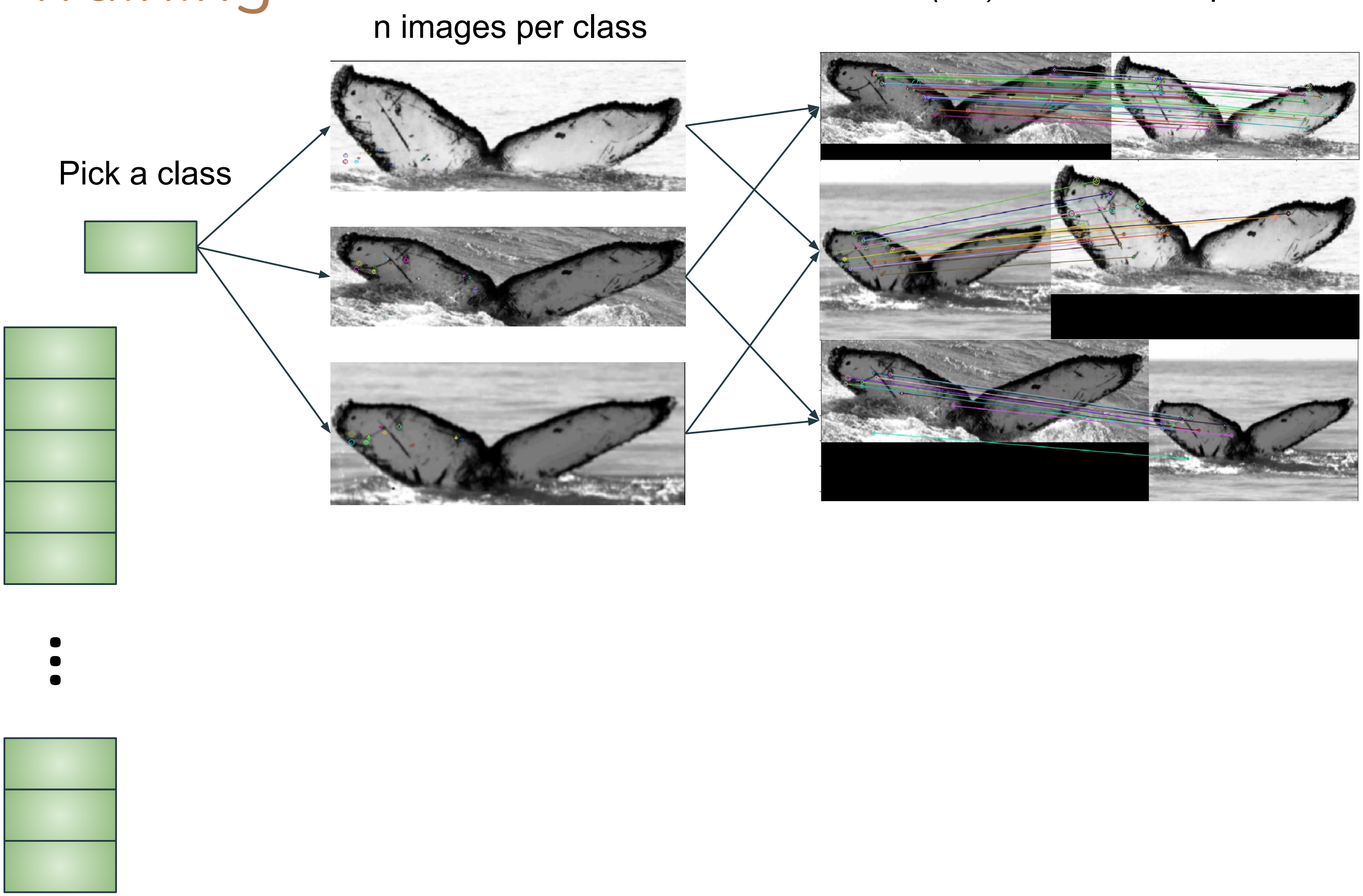
Pick a class



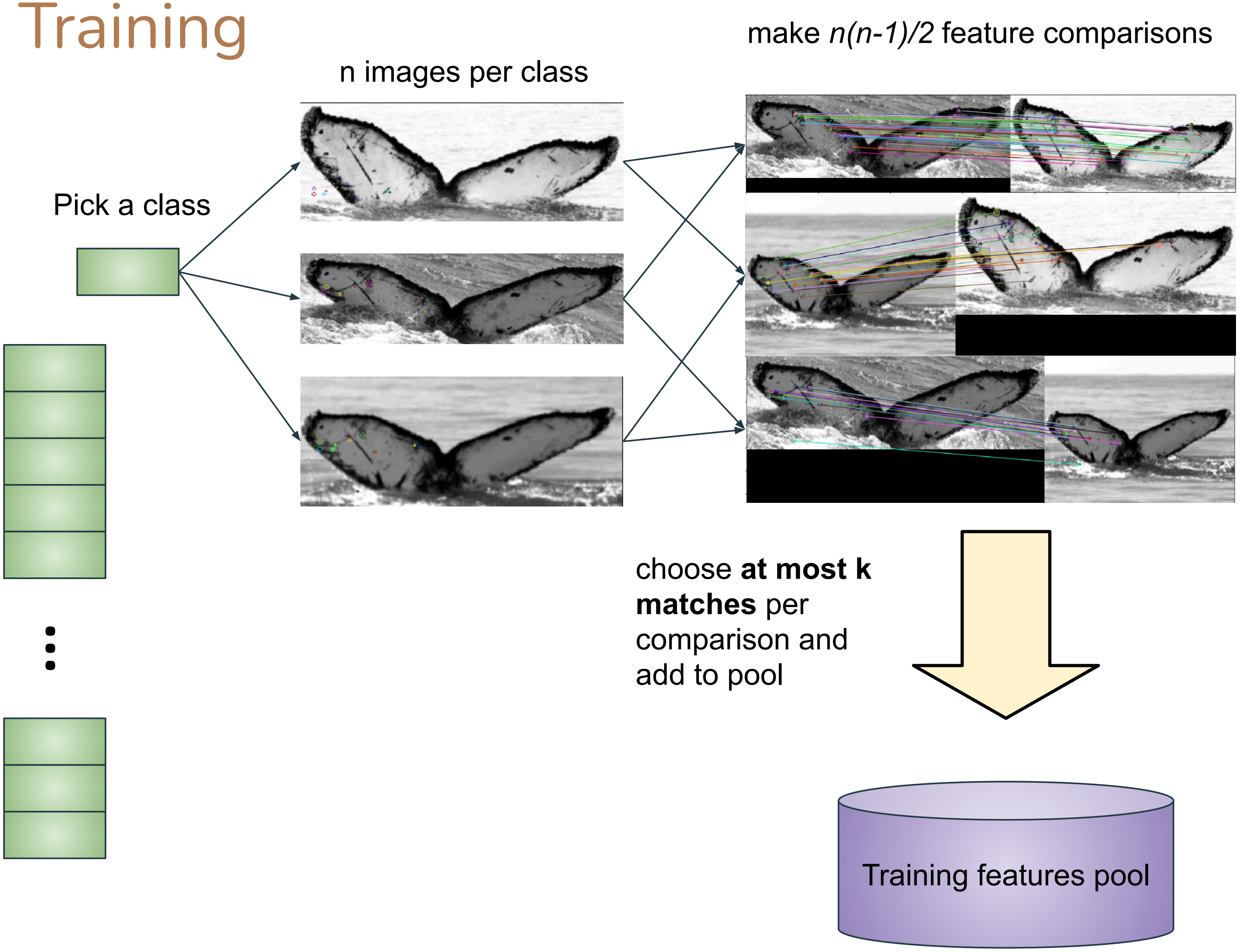
Training


n images per class
Pick a class
resized to same number of pixels

Training

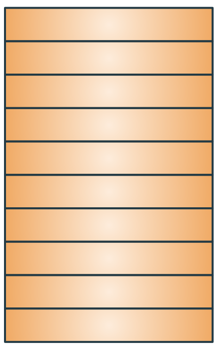

make $n(n-1)/2$ feature comparisons
n images per class
Pick a class

Training



make n(n-1)/2 feature comparisons
n images per class
Pick a class
choose **at most k matches** per comparison and add to pool
Training features pool

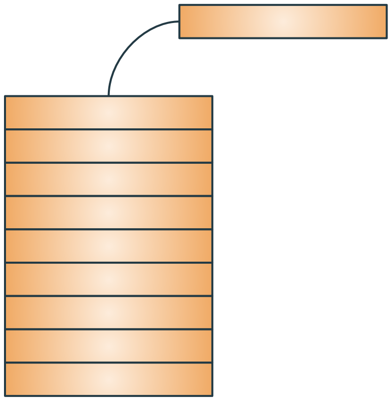
Testing



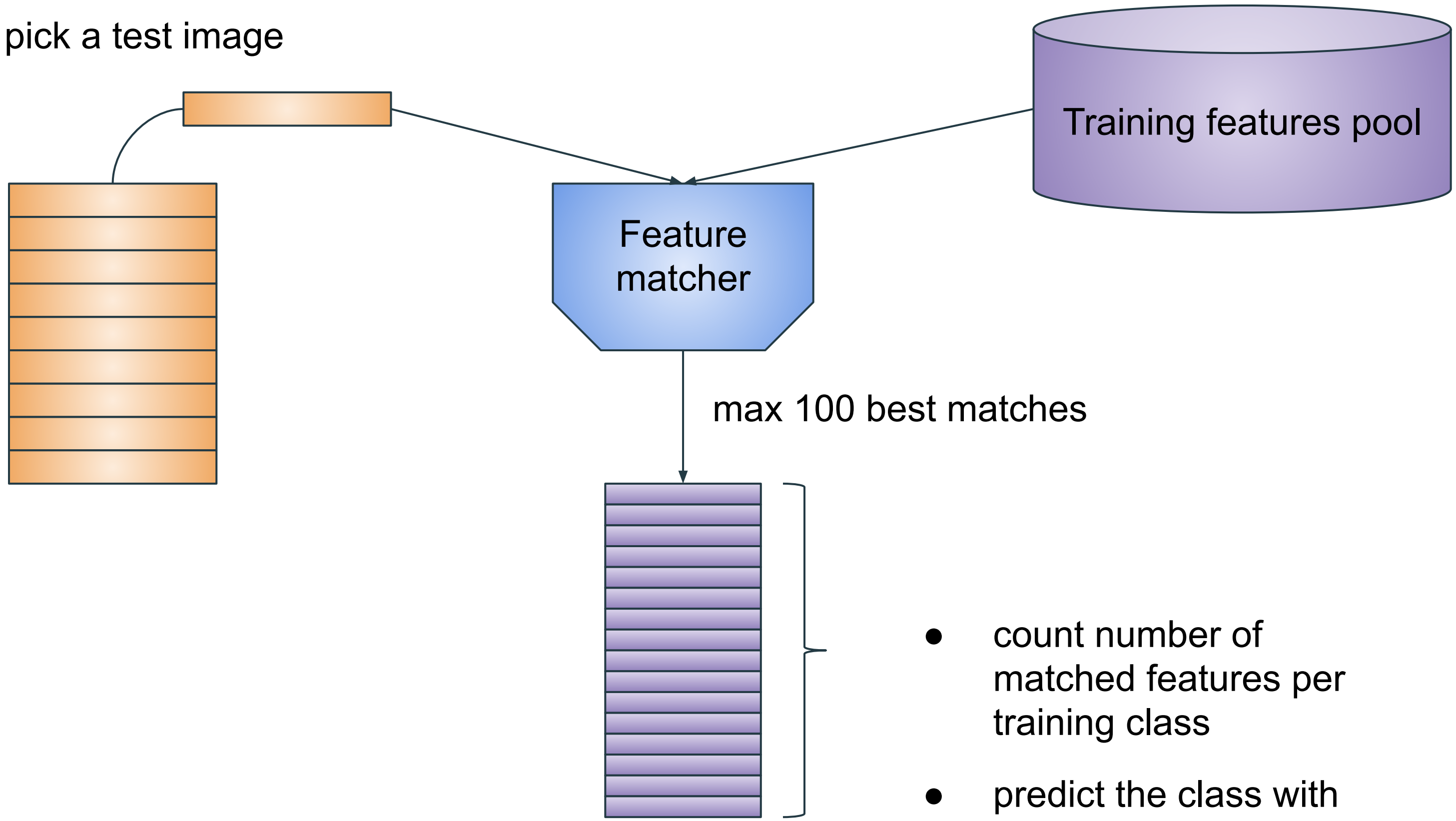
Test images

Testing

pick a test image



Testing



- count number of matched features per training class
- predict the class with highest number of matched components

Classification pipeline

Algorithm 1 Training for SIFT features

initialize $training\_features$
for each class c in training classes **do**
  n = number of training images in c
  for $i = 1 : n$ **do**
    for $j = i + 1 : n$ **do**
      $matched\_features$ = featureMatcher(i,j,$pixel\_threshold$)
      add ($matched\_features$,c) to $training\_features$
    end for
  end for
end for

Algorithm 2 Testing for SIFT features

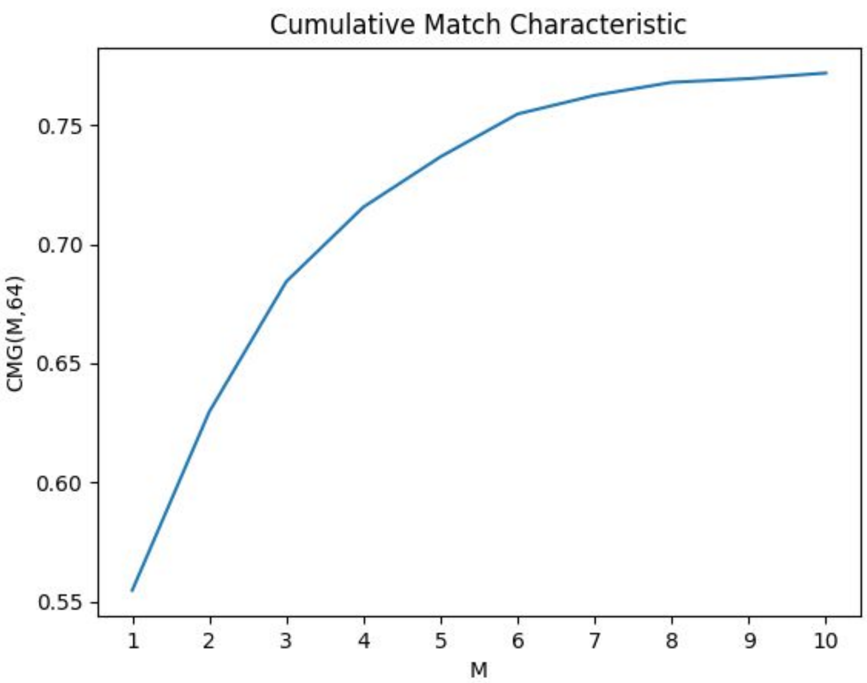
for t in test inputs **do**
  $matched\_features$ = featureMatcher(t, $training\_features$,$pixel\_threshold$)
  count class occurrences in $matched\_features$
  divide by number of features in $training\_features$ to remove bias
end for

Grid search

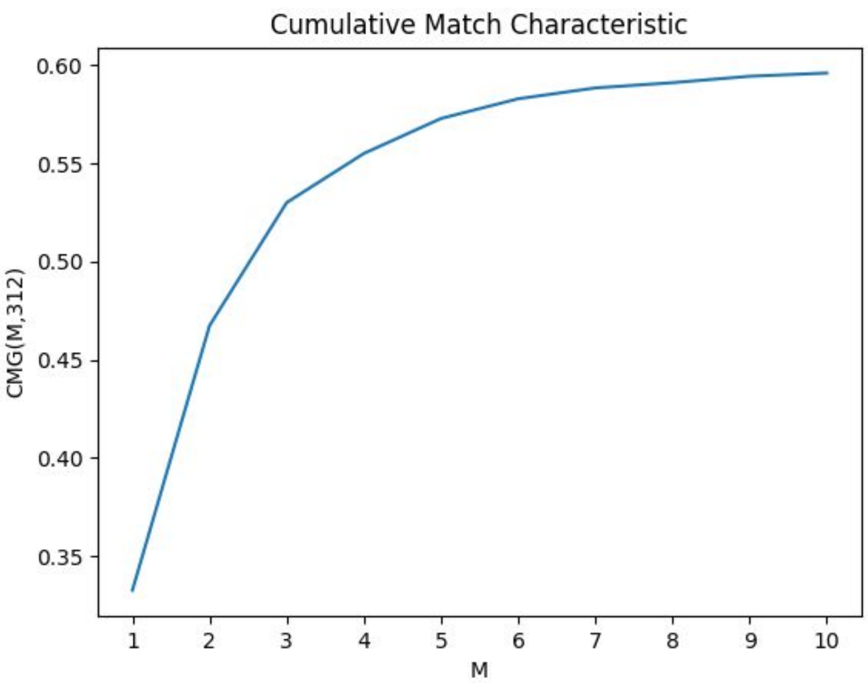
8 fold CV accuracies for pixels resized to 150,000

Pixel distance threshold	Maximum number of matches in feature matcher							
	20	**40**	**60**	**80**	**100**	**120**	**140**	**160**
100	53.4%	54.2%	53.8%	53.8%	54.0%	53.6%	53.9%	53.8%
110	53.7%	55.4%	55.1%	54.8%	55.1%	54.8%	55.1%	55.0%
120	54.7%	56.3%	55.8%	55.9%	56.2%	56.2%	56.1%	55.9%
150	54.6%	55.5%	55.9%	55.8%	55.8%	55.9%	56.2%	56.3%
200	54.6%	54.9%	54.5%	55.1%	55.3%	55.6%	55.7%	55.7%
250	55.5%	55.4%	55.3%	54.8%	55.0%	55.1%	55.2%	55.2%
300	54.0%	54.8%	54.0%	54.0%	53.8%	53.7%	53.7%	

Cumulative Match Characteristics for SIFT



64 class experiment



312 class experiment

Results

Average test accuracies for 10 splits

	64 class 512 images	312 class 1950 images
Eigen fluke	15.7%	7.6%
SIFT	55.4%	33.3%

Conclusion

- My SIFT prediction algorithm runs rather slowly and therefore parameter search takes very long time. Thus I could not optimize all parameters.
- Given that there are 312 classes, expected accuracy would be 0.3% if classes are randomly guessed.
- Even though I eliminated the pictures with bad enrollment, remaining images still preserved pose and scale variance, which possibly reduced performance.
- If I could have obtained probability for each class, for both methods, it would be possible to come up with a fusion scheme and increase rank-M accuracy.

Additional Materials

Data cleaning

- Dataset contained duplicate images, to eliminate those I followed the following kaggle kernel and used image hashing to detect replicate images
    - https://www.kaggle.com/stehai/duplicate-images-data-cleaning
- I also discarded the images that have bad enrollment

Eigen flukes - PCA algorithm

I used *Scikit-Learn* Python library for PCA computation. Their algorithm limits number of components so that

```
n_components == min(n_samples, n_features)
```

In my case, since the number of pixels is always greater than number of samples, this PCA algorithm found number of components that is equal to my training sample size.

I tried implementing PCA manually using SVD but my computer gave memory errors, therefore I sticked with built-in function.

Eigen flukes - KNN

I also tried k-nearest neighbour classifier with a few modifications:

- Used Euclidean distance
- For first k neighbours:
  - Find average distance of found classes
  - Divide it by the number of training samples for that class (for eliminating bias)

For example if k = 9 and we found neighbours that belong to classes A & B, with $k_A = 4$ and $k_B = 5$, with distances d_{Ai} and d_{Bi}. And also lets say class A has $n_A = 4$ training images and class has $n_B = 10$ training images. The scores are found as:

$$score(A) = \frac{\sum_{i=1}^{k_A} d_{A,i}}{k_A \times n_A}$$

I used this method to obtain confidence scores so that I could find cumulative match characteristics, but since it did not work well, I went for SVM and reported rank-1 accuracy.

Eigen flukes - KNN

KNN accuracies for 64 class experiment with 10-fold CV

k	Accuracy
1	6.2%
3	5.9%
5	5.8%
7	5.7%
9	6.0%
11	5.6%

SIFT

Homography test is applied during training feature pooling. I tried to utilize it for testing also, such that a test image would be compared to each class's features separately instead of matching to all training features. I tried using homography for testing too but obtained worse results, accuracies around 10% and it run slower. It might be due to erroneous implementation or some other factors.

SIFT

I used the same pixel threshold for both training and test feature matching.

I did not examine using different parameters for training and testing.