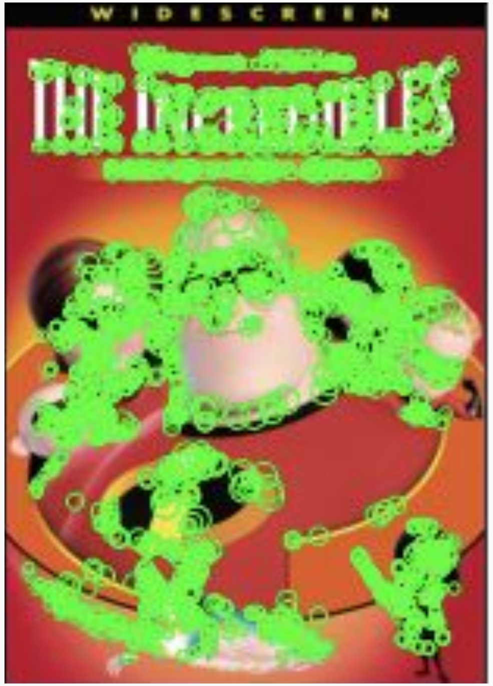
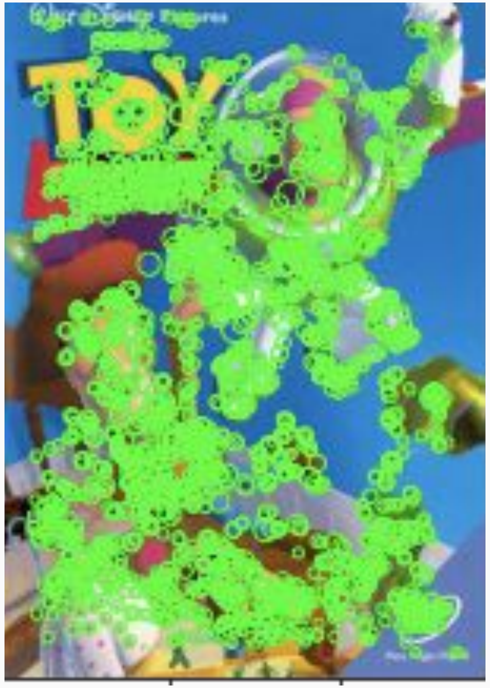
Implementing Visual Search with Vocabulary Trees

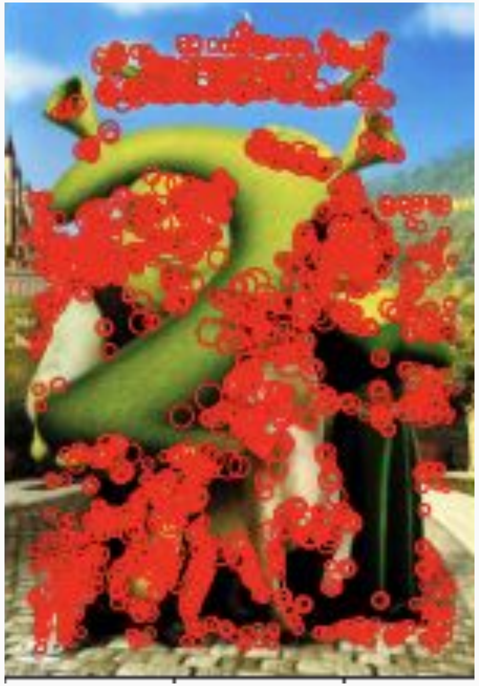
Mohamed Harmanani

Feature Extraction

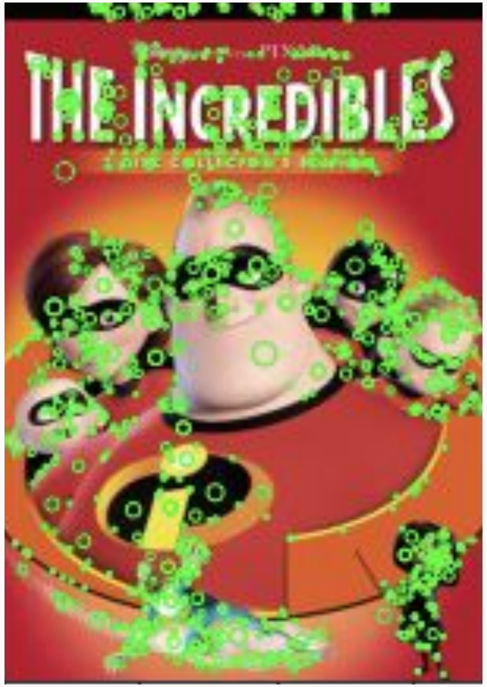
BRISK Detector

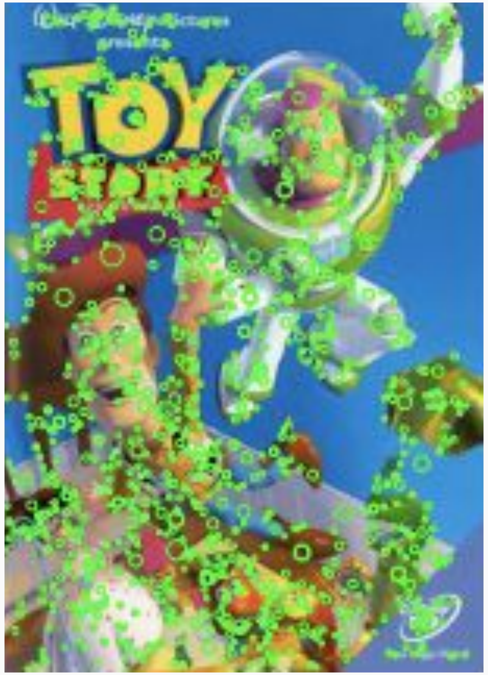


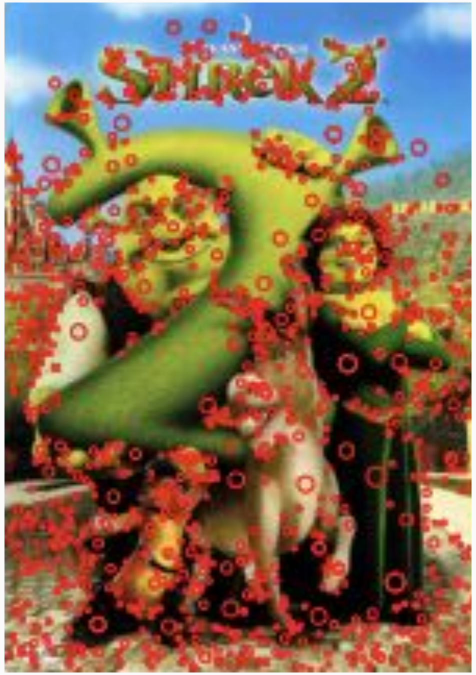




SIFT Detector

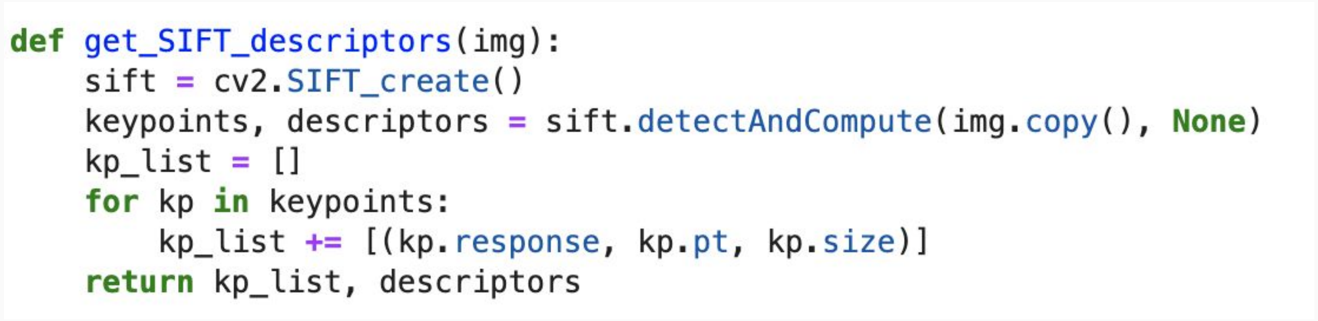






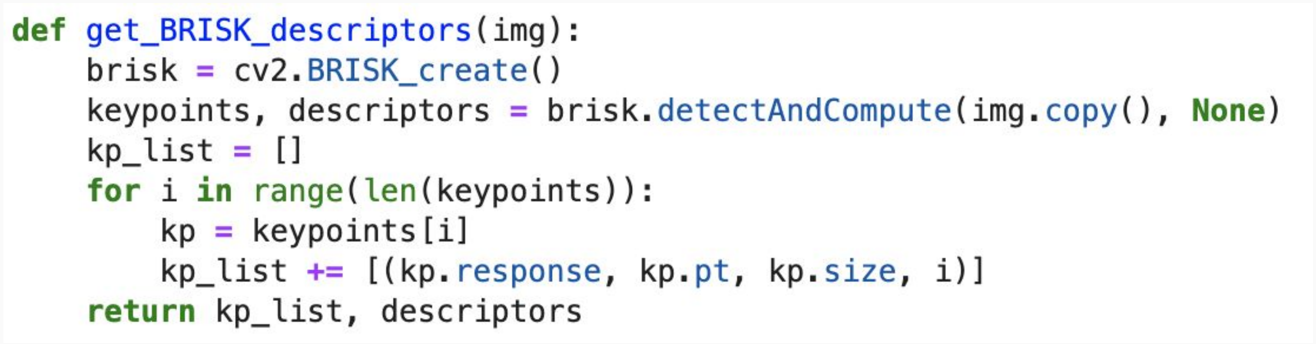
Code

Code to extract SIFT features



```python
def get_SIFT_descriptors(img):
    sift = cv2.SIFT_create()
    keypoints, descriptors = sift.detectAndCompute(img.copy(), None)
    kp_list = []
    for kp in keypoints:
        kp_list += [(kp.response, kp.pt, kp.size)]
    return kp_list, descriptors
```

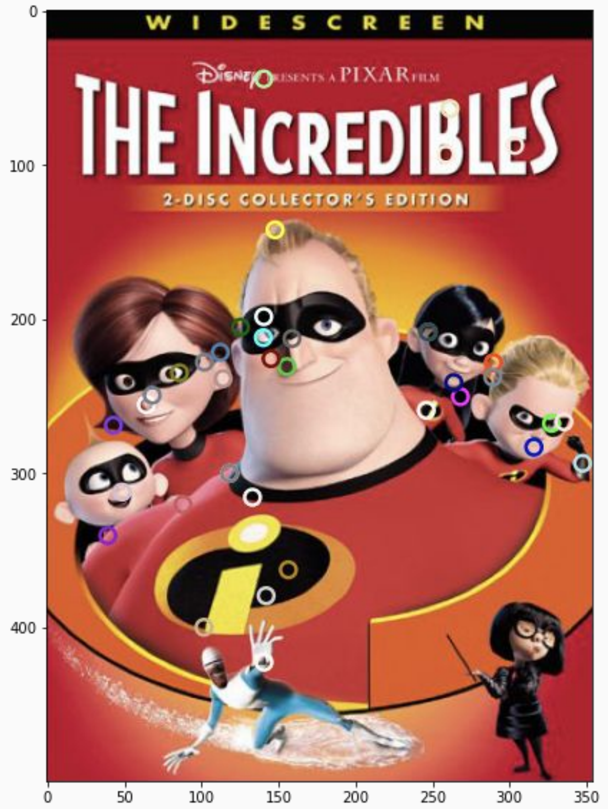
Code to extract BRISK features

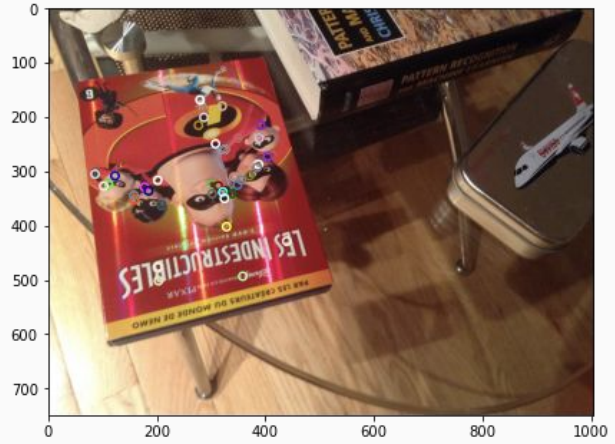


```python
def get_BRISK_descriptors(img):
    brisk = cv2.BRISK_create()
    keypoints, descriptors = brisk.detectAndCompute(img.copy(), None)
    kp_list = []
    for i in range(len(keypoints)):
        kp = keypoints[i]
        kp_list += [(kp.response, kp.pt, kp.size, i)]
    return kp_list, descriptors
```

Homography Mapping

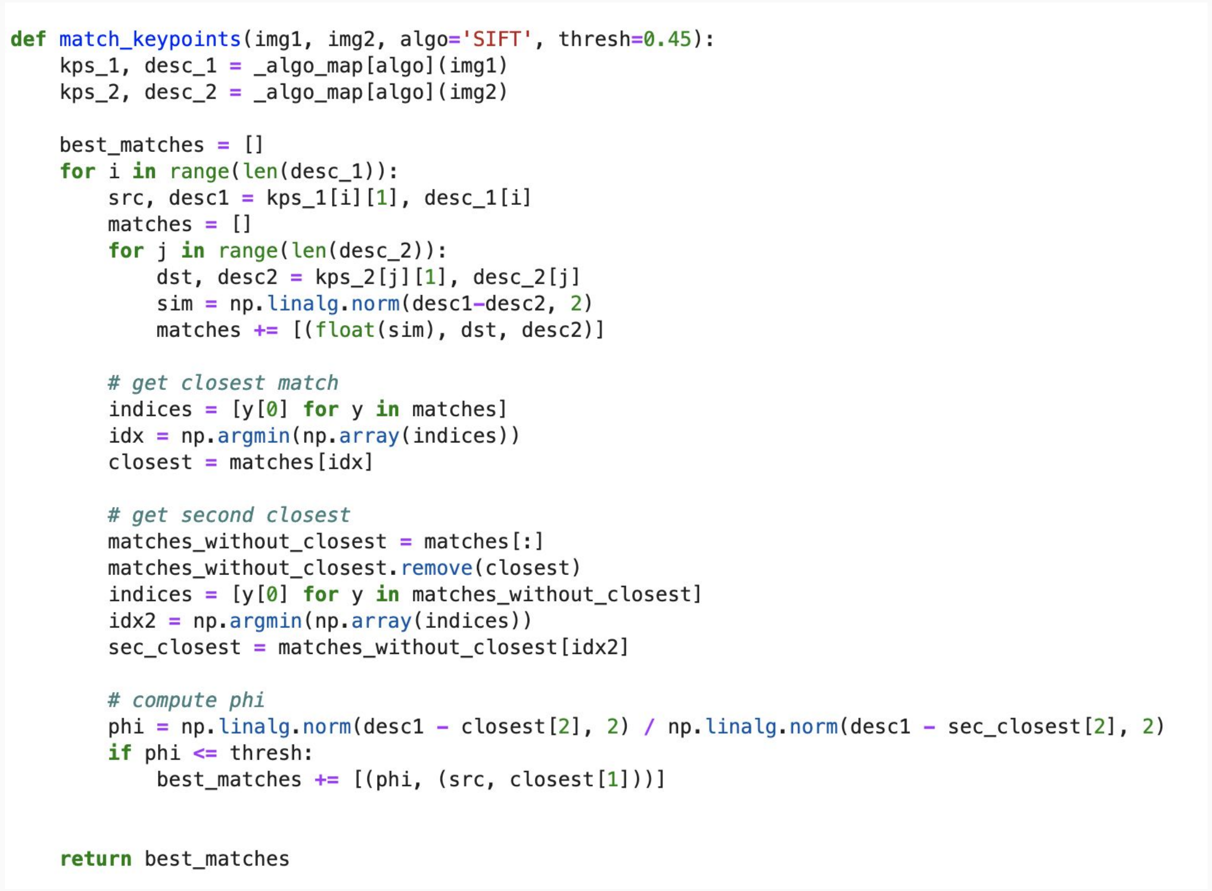
Step 1: Keypoint matching





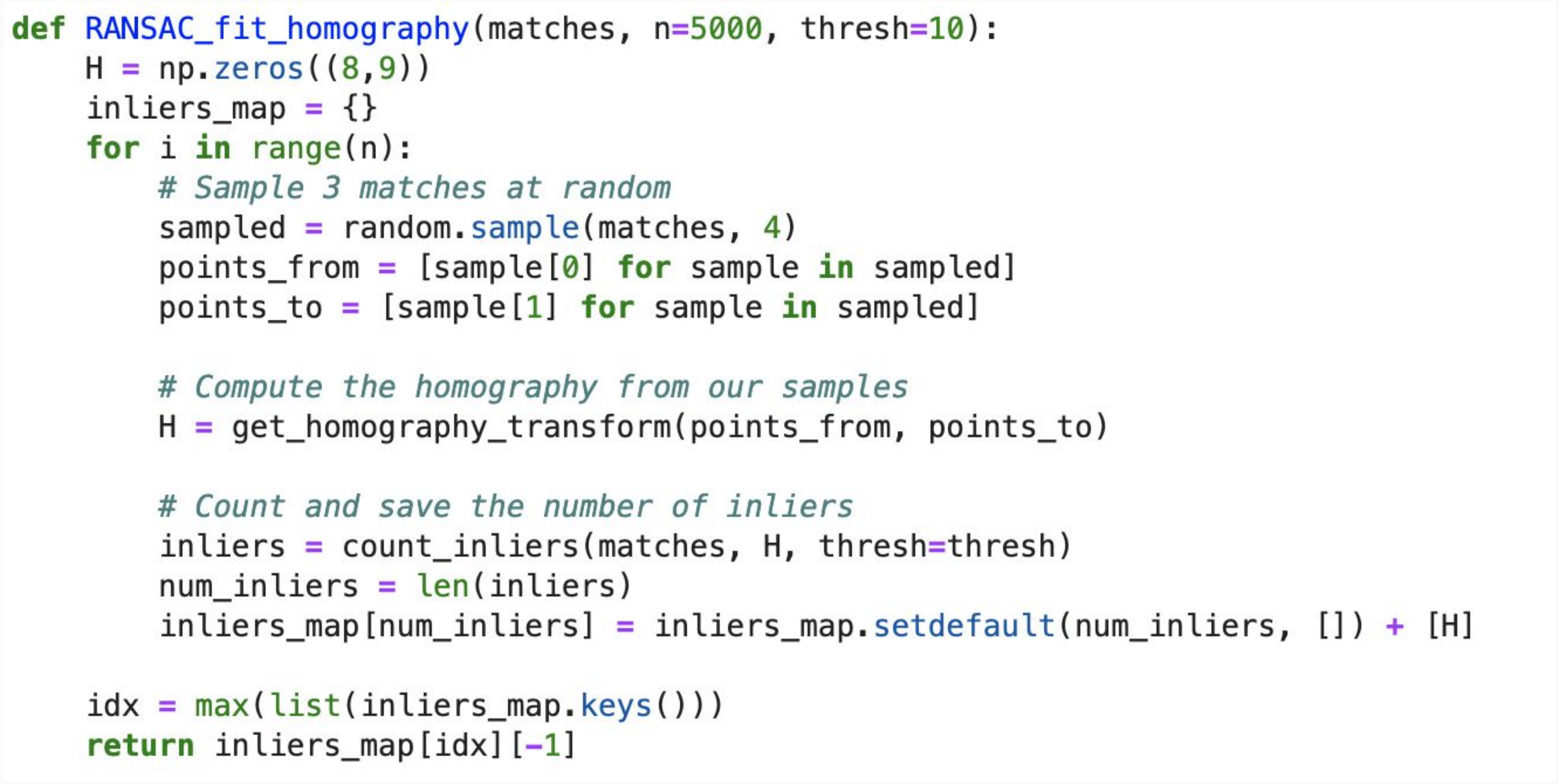
Using SIFT for keypoint matching

Code



```python
def match_keypoints(img1, img2, algo='SIFT', thresh=0.45):
    kps_1, desc_1 = _algo_map[algo](img1)
    kps_2, desc_2 = _algo_map[algo](img2)

    best_matches = []
    for i in range(len(desc_1)):
        src, desc1 = kps_1[i][1], desc_1[i]
        matches = []
        for j in range(len(desc_2)):
            dst, desc2 = kps_2[j][1], desc_2[j]
            sim = np.linalg.norm(desc1-desc2, 2)
            matches += [(float(sim), dst, desc2)]

        # get closest match
        indices = [y[0] for y in matches]
        idx = np.argmin(np.array(indices))
        closest = matches[idx]

        # get second closest
        matches_without_closest = matches[:]
        matches_without_closest.remove(closest)
        indices = [y[0] for y in matches_without_closest]
        idx2 = np.argmin(np.array(indices))
        sec_closest = matches_without_closest[idx2]

        # compute phi
        phi = np.linalg.norm(desc1 - closest[2], 2) / np.linalg.norm(desc1 - sec_closest[2], 2)
        if phi <= thresh:
            best_matches += [(phi, (src, closest[1]))]


    return best_matches
```

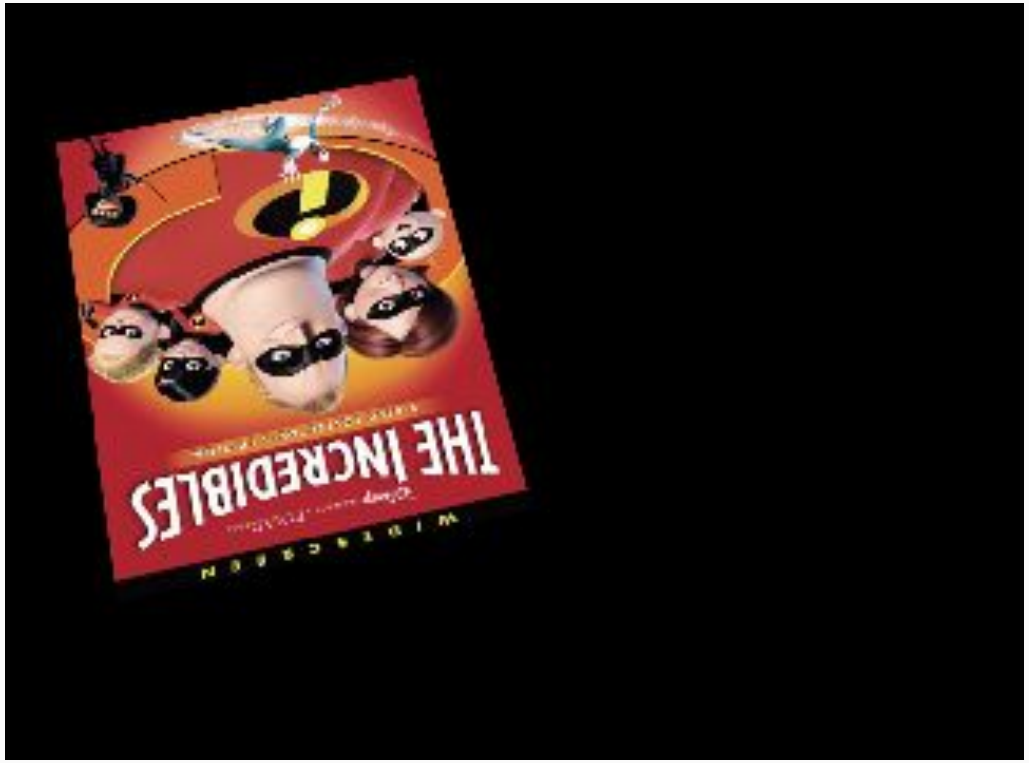
Step 2: RANSAC



```python
def RANSAC_fit_homography(matches, n=5000, thresh=10):
    H = np.zeros((8,9))
    inliers_map = {}
    for i in range(n):
        # Sample 3 matches at random
        sampled = random.sample(matches, 4)
        points_from = [sample[0] for sample in sampled]
        points_to = [sample[1] for sample in sampled]

        # Compute the homography from our samples
        H = get_homography_transform(points_from, points_to)

        # Count and save the number of inliers
        inliers = count_inliers(matches, H, thresh=thresh)
        num_inliers = len(inliers)
        inliers_map[num_inliers] = inliers_map.setdefault(num_inliers, []) + [H]

    idx = max(list(inliers_map.keys()))
    return inliers_map[idx][-1]
```

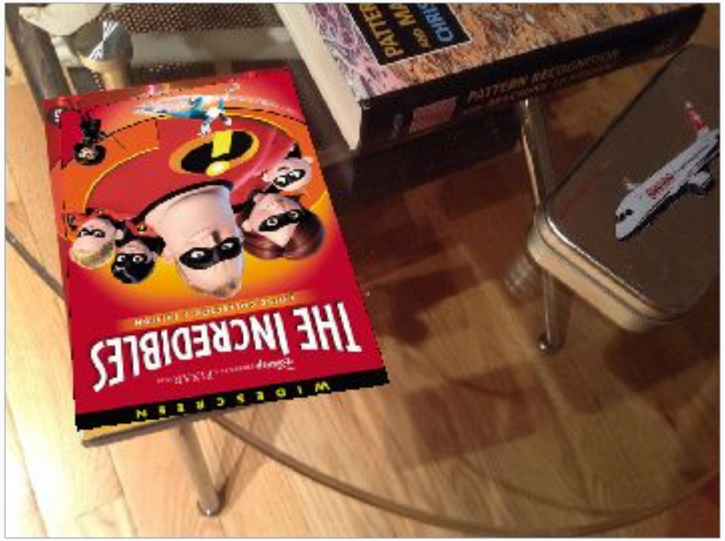
Python implementation of RANSAC

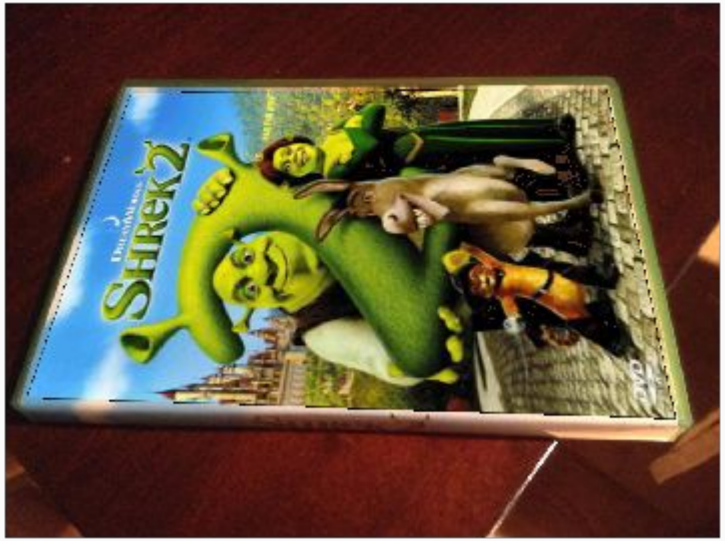
RANSAC Homography Result



Warped representation for example image

Step 3: Superpose Image





Superposing warped representations onto test images

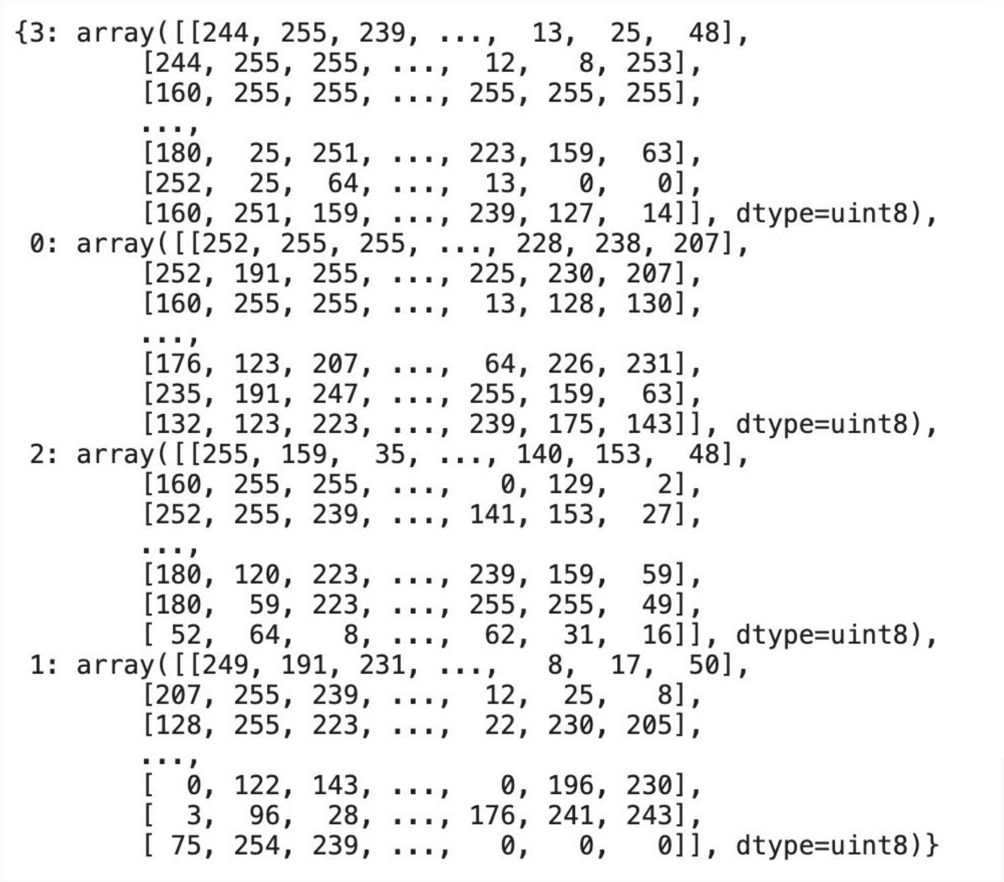
Efficient Index Retrieval

Building a feature database

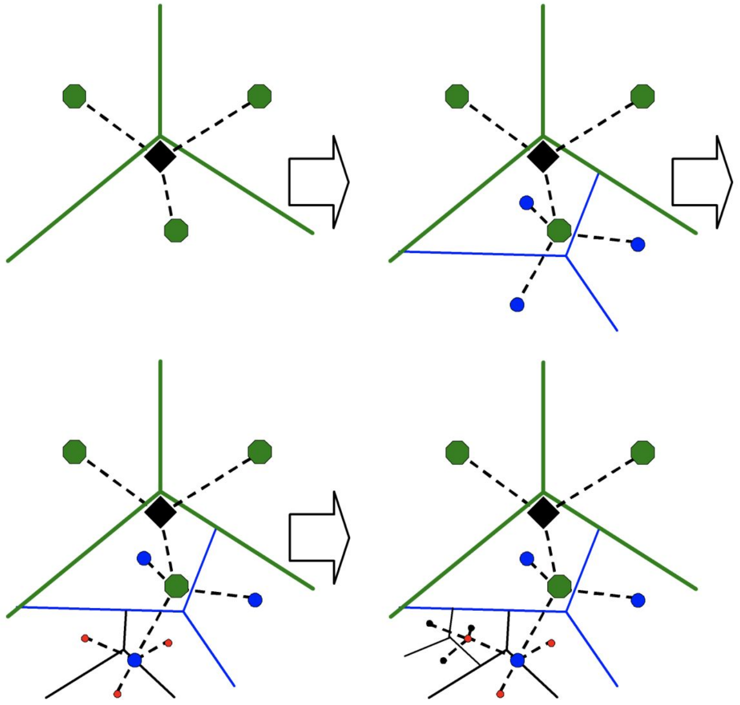
```
[244, 255, 239, ...,  13,  25,  48],
[244, 255, 255, ...,  12,   8, 253],
[252, 255, 255, ..., 228, 238, 207],
...,
[235, 191, 247, ..., 255, 159,  63],
[160, 251, 159, ..., 239, 127,  14],
[132, 123, 223, ..., 239, 175, 143]],
```

Finalized database of features

Partition the dataset using k-means

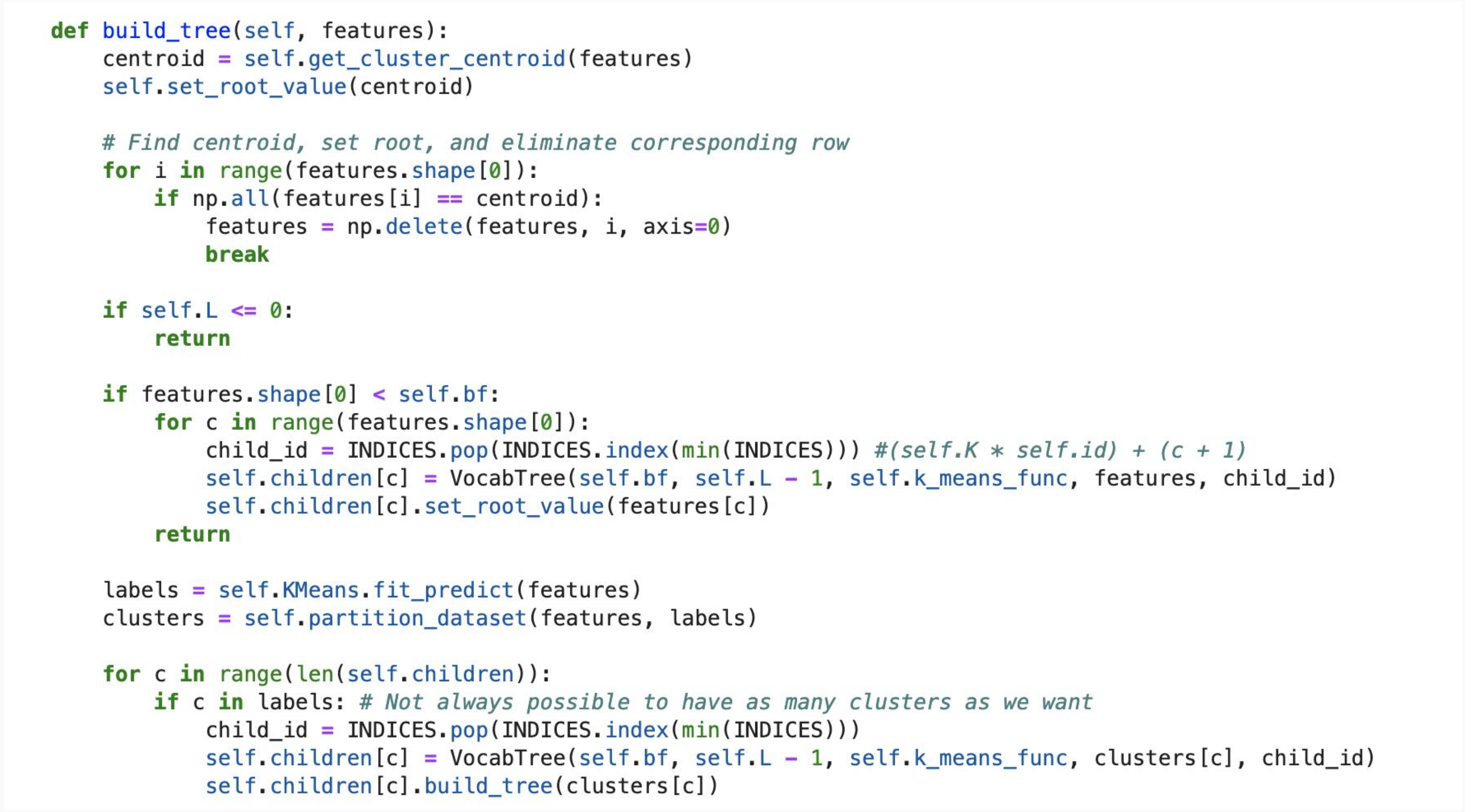


```
{3: array([[244, 255, 239, ...,  13,  25,  48],
        [244, 255, 255, ...,  12,   8, 253],
        [160, 255, 255, ..., 255, 255, 255],
        ...,
        [180,  25, 251, ..., 223, 159,  63],
        [252,  25,  64, ...,  13,   0,   0],
        [160, 251, 159, ..., 239, 127,  14]], dtype=uint8),
 0: array([[252, 255, 255, ..., 228, 238, 207],
        [252, 191, 255, ..., 225, 230, 207],
        [160, 255, 255, ...,  13, 128, 130],
        ...,
        [176, 123, 207, ...,  64, 226, 231],
        [235, 191, 247, ..., 255, 159,  63],
        [132, 123, 223, ..., 239, 175, 143]], dtype=uint8),
 2: array([[255, 159,  35, ..., 140, 153,  48],
        [160, 255, 255, ...,   0, 129,   2],
        [252, 255, 239, ..., 141, 153,  27],
        ...,
        [180, 120, 223, ..., 239, 159,  59],
        [180,  59, 223, ..., 255, 255,  49],
        [ 52,  64,   8, ...,  62,  31,  16]], dtype=uint8),
 1: array([[249, 191, 231, ...,   8,  17,  50],
        [207, 255, 239, ...,  12,  25,   8],
        [128, 255, 223, ...,  22, 230, 205],
        ...,
        [  0, 122, 143, ...,   0, 196, 230],
        [  3,  96,  28, ..., 176, 241, 243],
        [ 75, 254, 239, ...,   0,   0,   0]], dtype=uint8)}
```



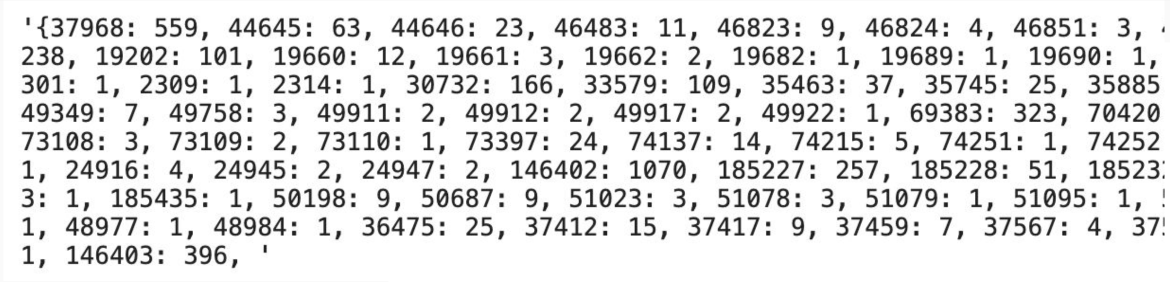
[Source: Nister and Stewenius 2006]

Code



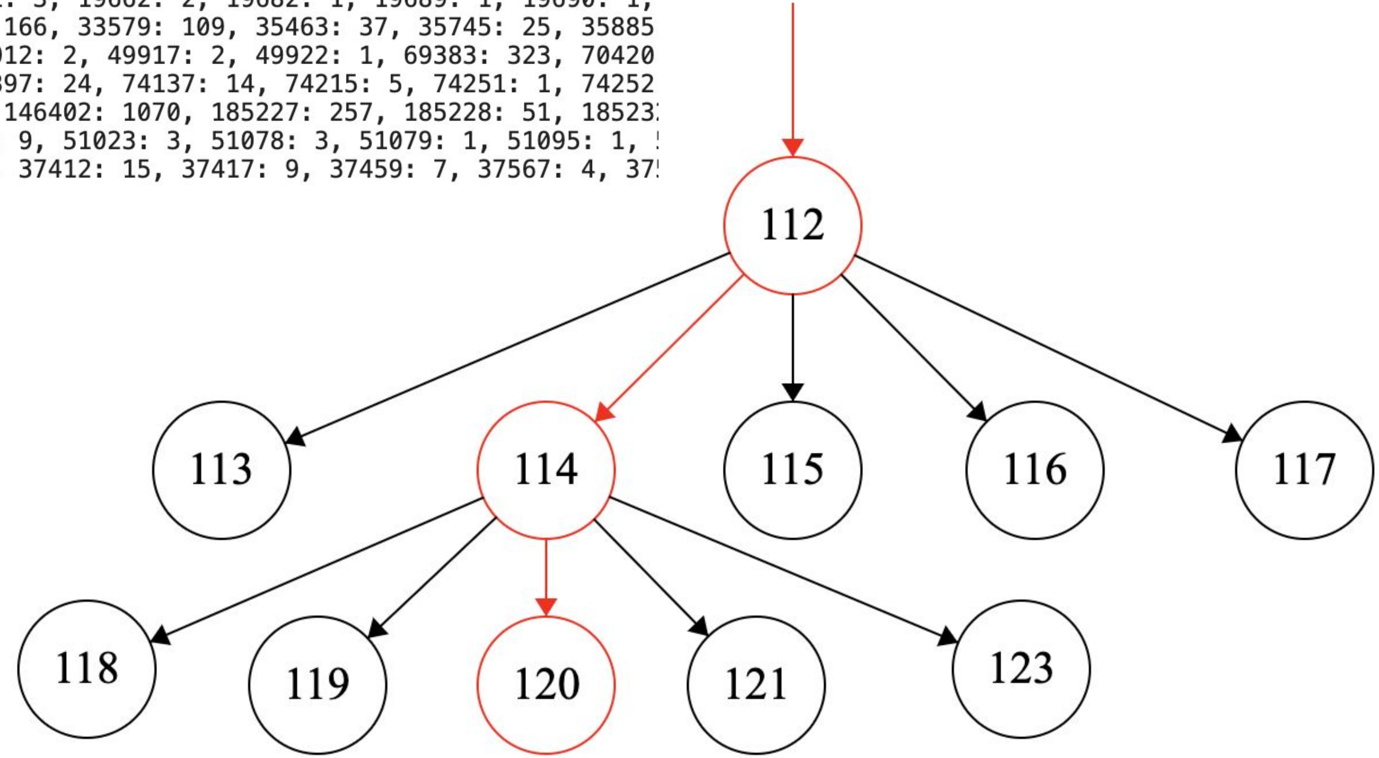
```python
def build_tree(self, features):
    centroid = self.get_cluster_centroid(features)
    self.set_root_value(centroid)

    # Find centroid, set root, and eliminate corresponding row
    for i in range(features.shape[0]):
        if np.all(features[i] == centroid):
            features = np.delete(features, i, axis=0)
            break

    if self.L <= 0:
        return

    if features.shape[0] < self.bf:
        for c in range(features.shape[0]):
            child_id = INDICES.pop(INDICES.index(min(INDICES))) #(self.K * self.id) + (c + 1)
            self.children[c] = VocabTree(self.bf, self.L - 1, self.k_means_func, features, child_id)
            self.children[c].set_root_value(features[c])
        return

    labels = self.KMeans.fit_predict(features)
    clusters = self.partition_dataset(features, labels)

    for c in range(len(self.children)):
        if c in labels: # Not always possible to have as many clusters as we want
            child_id = INDICES.pop(INDICES.index(min(INDICES)))
            self.children[c] = VocabTree(self.bf, self.L - 1, self.k_means_func, clusters[c], child_id)
            self.children[c].build_tree(clusters[c])
```

Propagate features down the vocabulary tree

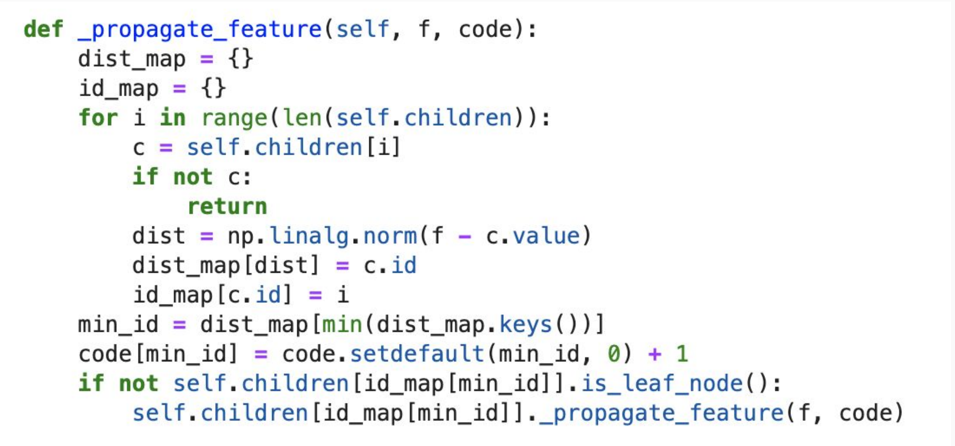


```
'{37968: 559, 44645: 63, 44646: 23, 46483: 11, 46823: 9, 46824: 4, 46851: 3,
238, 19202: 101, 19660: 12, 19661: 3, 19662: 2, 19682: 1, 19689: 1, 19690: 1,
301: 1, 2309: 1, 2314: 1, 30732: 166, 33579: 109, 35463: 37, 35745: 25, 35885
49349: 7, 49758: 3, 49911: 2, 49912: 2, 49917: 2, 49922: 1, 69383: 323, 70420
73108: 3, 73109: 2, 73110: 1, 73397: 24, 74137: 14, 74215: 5, 74251: 1, 74252
1, 24916: 4, 24945: 2, 24947: 2, 146402: 1070, 185227: 257, 185228: 51, 18523
3: 1, 185435: 1, 50198: 9, 50687: 9, 51023: 3, 51078: 3, 51079: 1, 51095: 1,
1, 48977: 1, 48984: 1, 36475: 25, 37412: 15, 37417: 9, 37459: 7, 37567: 4, 37
1, 146403: 396, '
```

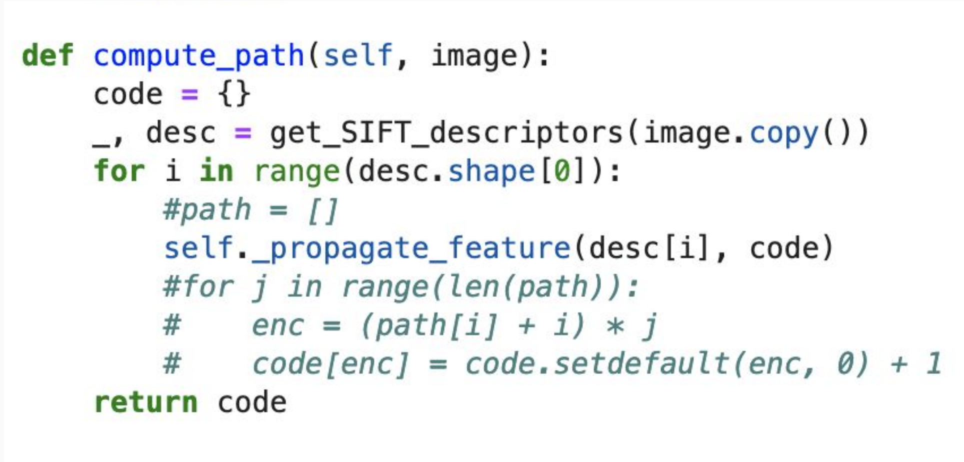
Bag of features



Code

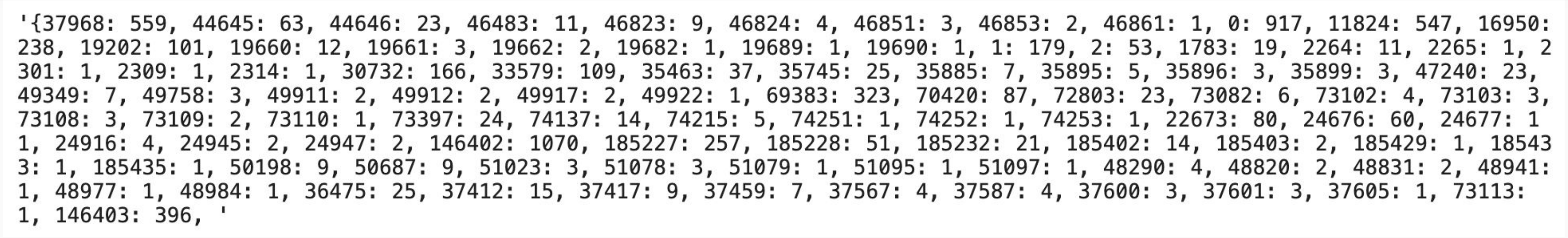


```python
def _propagate_feature(self, f, code):
    dist_map = {}
    id_map = {}
    for i in range(len(self.children)):
        c = self.children[i]
        if not c:
            return
        dist = np.linalg.norm(f - c.value)
        dist_map[dist] = c.id
        id_map[c.id] = i
    min_id = dist_map[min(dist_map.keys())]
    code[min_id] = code.setdefault(min_id, 0) + 1
    if not self.children[id_map[min_id]].is_leaf_node():
        self.children[id_map[min_id]]._propagate_feature(f, code)
```



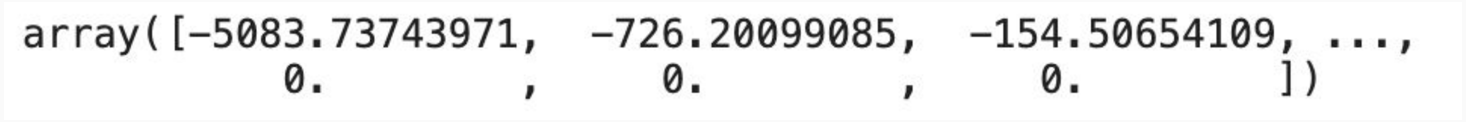
```python
def compute_path(self, image):
    code = {}
    _, desc = get_SIFT_descriptors(image.copy())
    for i in range(desc.shape[0]):
        #path = []
        self._propagate_feature(desc[i], code)
        #for j in range(len(path)):
        #    enc = (path[i] + i) * j
        #    code[enc] = code.setdefault(enc, 0) + 1
    return code
```

tf-idf weighting



```
'{37968: 559, 44645: 63, 44646: 23, 46483: 11, 46823: 9, 46824: 4, 46851: 3, 46853: 2, 46861: 1, 0: 917, 11824: 547, 16950:
238, 19202: 101, 19660: 12, 19661: 3, 19662: 2, 19682: 1, 19689: 1, 19690: 1, 1: 179, 2: 53, 1783: 19, 2264: 11, 2265: 1, 2
301: 1, 2309: 1, 2314: 1, 30732: 166, 33579: 109, 35463: 37, 35745: 25, 35885: 7, 35895: 5, 35896: 3, 35899: 3, 47240: 23,
49349: 7, 49758: 3, 49911: 2, 49912: 2, 49917: 2, 49922: 1, 69383: 323, 70420: 87, 72803: 23, 73082: 6, 73102: 4, 73103: 3,
73108: 3, 73109: 2, 73110: 1, 73397: 24, 74137: 14, 74215: 5, 74251: 1, 74252: 1, 74253: 1, 22673: 80, 24676: 60, 24677: 1
1, 24916: 4, 24945: 2, 24947: 2, 146402: 1070, 185227: 257, 185228: 51, 185232: 21, 185402: 14, 185403: 2, 185429: 1, 18543
3: 1, 185435: 1, 50198: 9, 50687: 9, 51023: 3, 51078: 3, 51079: 1, 51095: 1, 51097: 1, 48290: 4, 48820: 2, 48831: 2, 48941:
1, 48977: 1, 48984: 1, 36475: 25, 37412: 15, 37417: 9, 37459: 7, 37567: 4, 37587: 4, 37600: 3, 37601: 3, 37605: 1, 73113:
1, 146403: 396, '
```

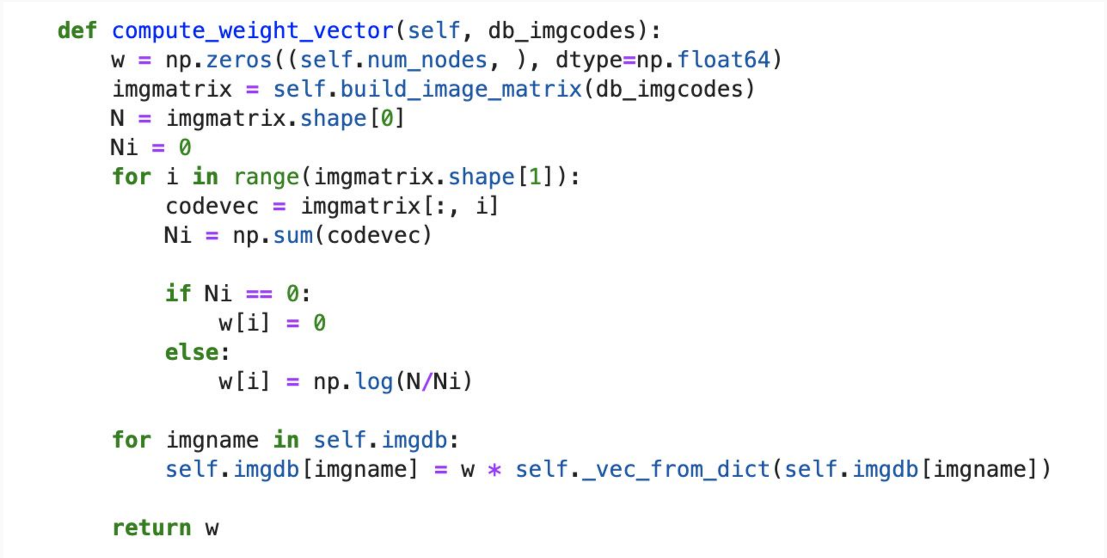
Bag of features



```
array([-5083.73743971,  -726.20099085,  -154.50654109, ...,
           0.        ,     0.        ,     0.        ])
```

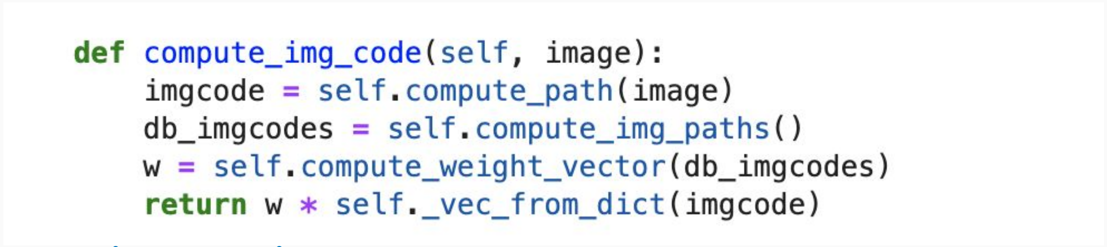
Finalized tf-idf representation

Code



```python
def compute_weight_vector(self, db_imgcodes):
    w = np.zeros((self.num_nodes, ), dtype=np.float64)
    imgmatrix = self.build_image_matrix(db_imgcodes)
    N = imgmatrix.shape[0]
    Ni = 0
    for i in range(imgmatrix.shape[1]):
        codevec = imgmatrix[:, i]
        Ni = np.sum(codevec)

        if Ni == 0:
            w[i] = 0
        else:
            w[i] = np.log(N/Ni)

    for imgname in self.imgdb:
        self.imgdb[imgname] = w * self._vec_from_dict(self.imgdb[imgname])

    return w
```

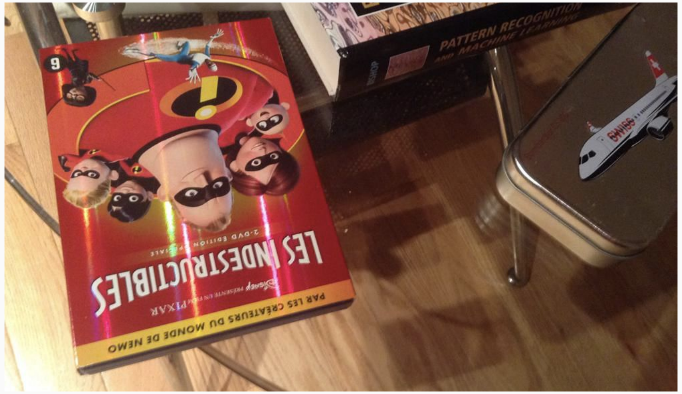
Computing the weights

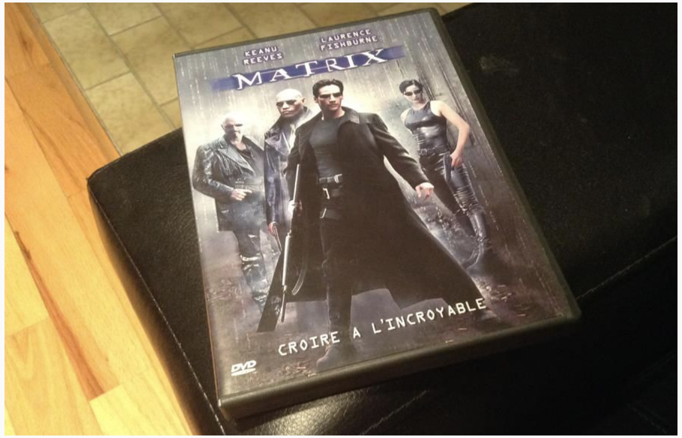


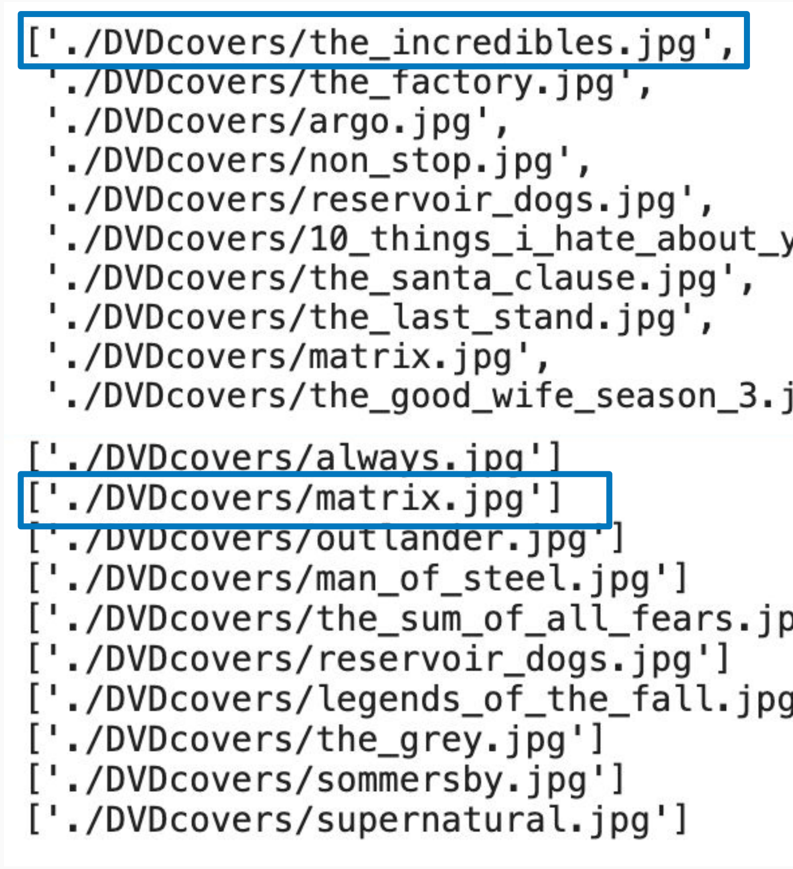
```python
def compute_img_code(self, image):
    imgcode = self.compute_path(image)
    db_imgcodes = self.compute_img_paths()
    w = self.compute_weight_vector(db_imgcodes)
    return w * self._vec_from_dict(imgcode)
```

Applying the weights

Retrieving matches


PATTERN RECOGNITION AND MACHINE LEARNING
LES INDESTRUCTIBLES
PIXAR
PAR LES CRÉATEURS DU MONDE DE NEMO

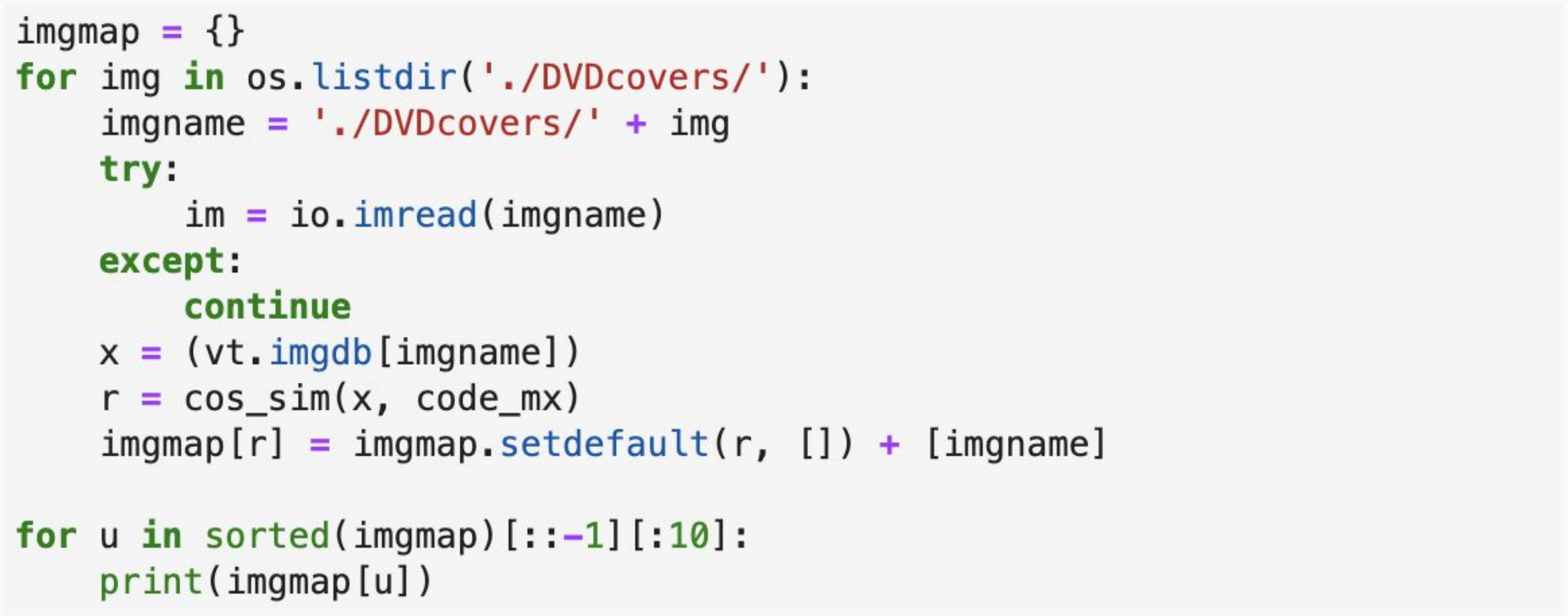




```
['./DVDcovers/the_incredibles.jpg',
 './DVDcovers/the_factory.jpg',
 './DVDcovers/argo.jpg',
 './DVDcovers/non_stop.jpg',
 './DVDcovers/reservoir_dogs.jpg',
 './DVDcovers/10_things_i_hate_about_y
 './DVDcovers/the_santa_clause.jpg',
 './DVDcovers/the_last_stand.jpg',
 './DVDcovers/matrix.jpg',
 './DVDcovers/the_good_wife_season_3.j
```

```
['./DVDcovers/always.jpg']
['./DVDcovers/matrix.jpg']
['./DVDcovers/outlander.jpg']
['./DVDcovers/man_of_steel.jpg']
['./DVDcovers/the_sum_of_all_fears.jp
['./DVDcovers/reservoir_dogs.jpg']
['./DVDcovers/legends_of_the_fall.jpg
['./DVDcovers/the_grey.jpg']
['./DVDcovers/sommersby.jpg']
['./DVDcovers/supernatural.jpg']
```

Code



```
imgmap = {}
for img in os.listdir('./DVDcovers/'):
    imgname = './DVDcovers/' + img
    try:
        im = io.imread(imgname)
    except:
        continue
    x = (vt.imgdb[imgname])
    r = cos_sim(x, code_mx)
    imgmap[r] = imgmap.setdefault(r, []) + [imgname]

for u in sorted(imgmap)[::-1][:10]:
    print(imgmap[u])
```

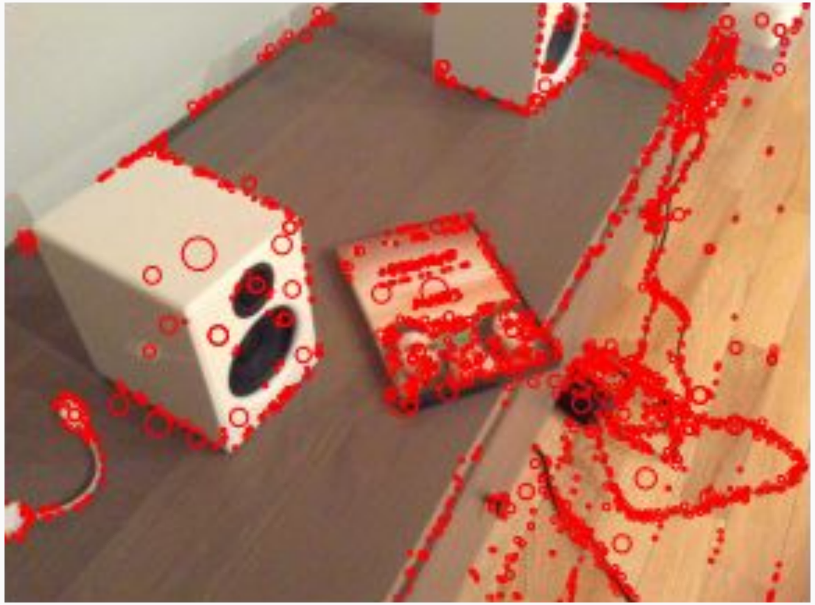
Code for printing top 10 retrieved matches

Issues & Discussion

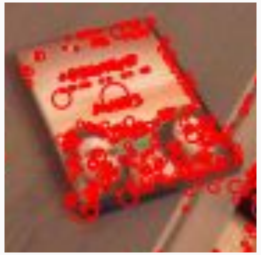
Runtime complexity

$$O(K^L)$$

Problem Images

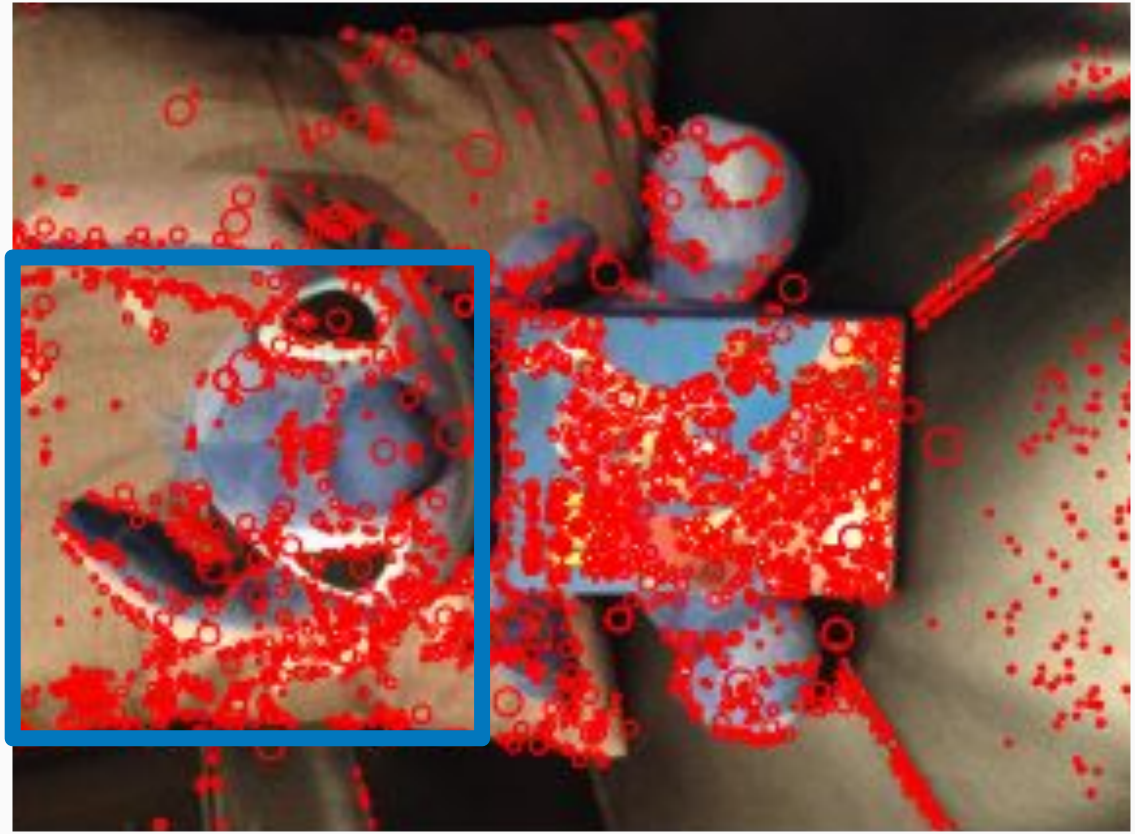


Score: ~0.79



Score: ~0.94

Problem Images



Excessive Noise

>>>