《 机 器 视 觉 》

实

验

报

告

学号： 2018213106

姓名： 刘嘉伟

班级： 计算机创新实验18-1班

# 实验一

## 实验目的

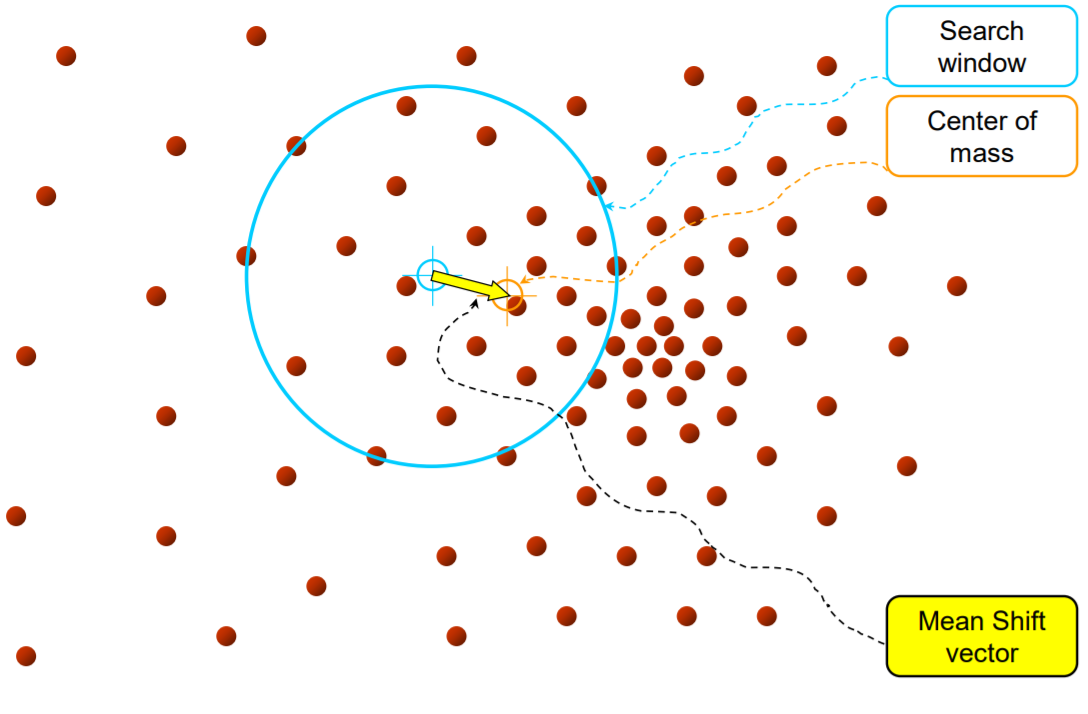
实现图像分割算法，并对图像分割算法的的效果进行分析与评价

## 实验内容

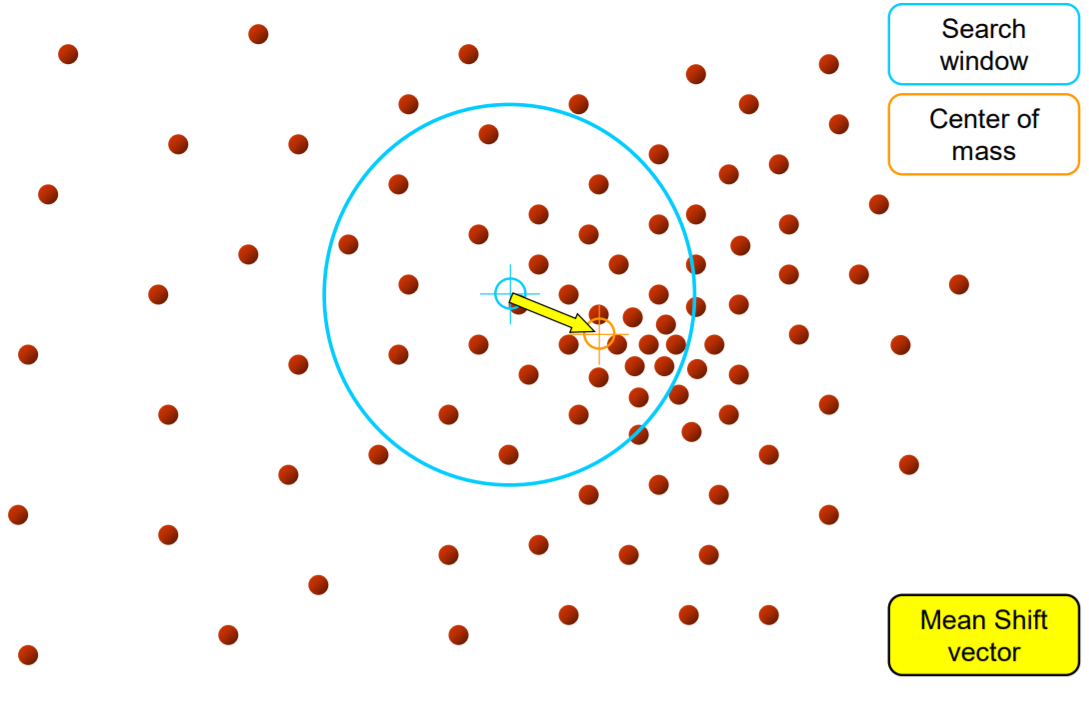
实现了mean-shift图像分割算法

## Mean-Shift算法原理

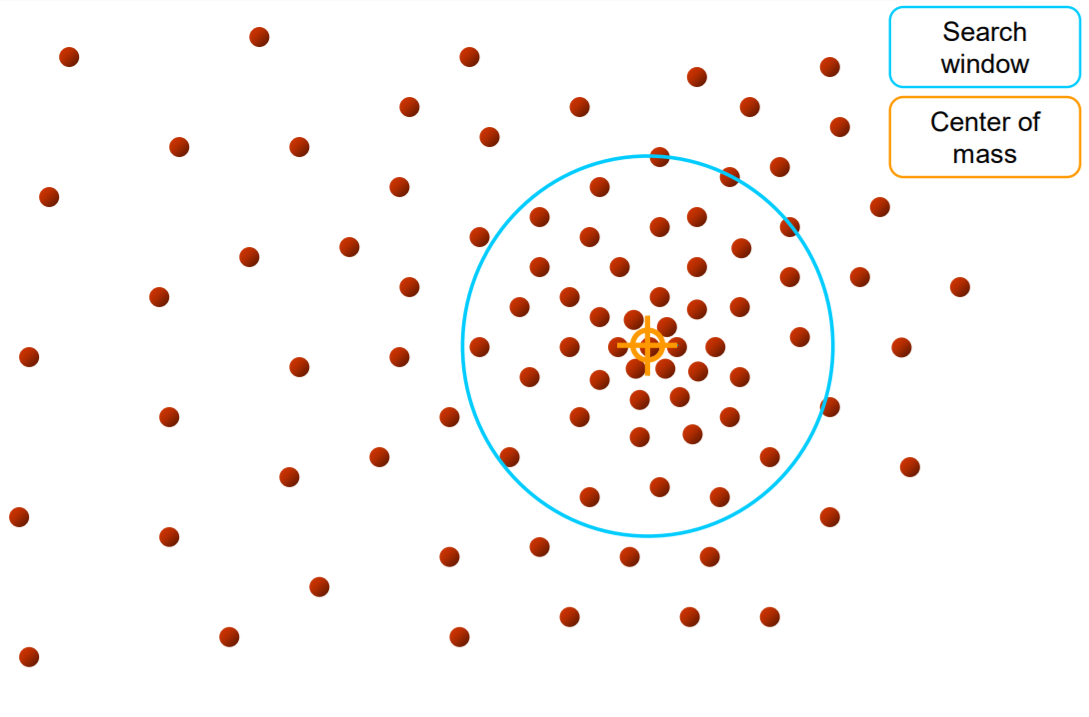
Mean-shift算法，中文译作均值漂移算法，其核心思想就是，假如在一个二维平面上分布着许多的点，每一个区域的点的密度都不一样，如下图所示：



均值漂移算法是对图像上的每个点进行如下处理，假设当前处理的点是：选定一个搜索半径R，对距离周围R范围内的点，计算所有点的平均位置向量，，其中k为半径R范围内点的数量，然后将点漂移至平均位置，如下图：



然后重复上述操作直至收敛，最终结果如下：

****

以上便是mean-shift算法的基本原理，但是如何将该算法用于图像分割呢？下面我们将该算法应用到图像分割中，变成一个图像分割算法。

## 四、Mean-Shift图像分割算法

图像中的点包括两类信息：坐标空间（spatial，，），颜色空间（range ，，）。这些就构成了特征空间。

**模点搜索**：某一个点，它在联合特征空间中迭代搜索它的mode/模点；

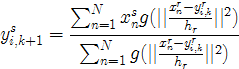
**图像平滑:**将模点的颜色值赋给它自己，即.对应原论文文中的**图像平滑，**实质上是**通过模点搜索，达到图像平滑的效果**

设点依次爬过的脚印为：

出发时，它所收敛到的模点为，c代表convergence。

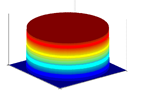
**第一步**：如果迭代次数超过最大值（默认最多爬最多100次），结束搜索跳到第四步，否则，在坐标空间，筛选靠近的数据点进入下一步计算。

**第二步**：使用第一步幸存下来的点计算重心，并向重心移动。



其中是某种核函数，比如高斯分布， 是颜色空间的核平滑尺度。OpenCV内置函数使用的是最简单的均匀分布：





二维可视化效果

是一个以（第步位置的颜色值）为球心，半径为的球体，球体内部值为1，球体外部值为0。对于经过上一步筛选后幸存的数据点，如果其颜色值满足，也就是颜色值落也在球内，那么求重心时，就要算上，否则落在球外，算重心时，就不带上它。实际上，**上一步****是依据坐标空间距离筛选数据点，是依据颜色距离进一步筛选数据点，上一步的筛子是圆形，这一步是球体**。

简而言之，设满足的点依次为，那么重心计算公式可以进一步化简为：



**注意**：上文中的两个参数，是Mean shift最核心的两个参数，具有直观的意义，分别代表坐标空间和颜色空间的核函数带宽。

第三步：判断是否到模点了，到了就停止。

如果，移动后颜色或者位置变化很小，则结束搜索，跳到第四步，否则重返第一步，从继续爬。

OpenCV内置函数停止搜索的条件：

（1）坐标距离不变

（2）颜色变化值很小。

满足一条就可以功成身退，否则继续努力。

第四步：将模点的颜色赋给出发点/，即。

## 五、运行Mean-Shift算法后的处理步骤

在使用Mean-Shift算法处理图像后，我们已经将图像的每个像素点进行了均值漂移，但是这并不足以完成我们需要的图像分割达到的效果，我们需要进行进一步处理，这里我使用的是**合并相似区域/模点聚类**算法，其核心思想就是，从图像上一个点开始，如果和它附近的点（4/8邻域）的颜色值相似就合并，同时再从新合并的点出发继续合并下去，直到碰到不相似的点或者该点已经属于另一类了，此时，就退回来，直到退无可退（所有的4/8邻域搜索空间都已经搜索完毕）。

在这步处理结果后，还可能有一些小区域没有被合并起来，可能会影响效果，所以还可以选择性的进行**兼并小区域**，从一没有合并到大区域的小区域出发，寻找最近的大区域，将其像素值合并到大区域中。

## 六、算法的C++实现

超参数定义如下：

|  |
| --- |
| String image\_path = "L:\\software\\OpenCV\\opencv\\sources\\samples\\data\\elephant.jpg"; //输入图像位置  int hr = 16; //距离尺度  int hs = 32; //颜色尺度  int iter = 100; //最大迭代次数  int blur\_size = 3; //模糊卷积核大小  //GaussianBlur、medianBlur  String blur\_mode = "GaussianBlur"; //模糊方法选择  int ColorThreshold = 10; //停止迭代时 最小变化像素差值  int merge\_mode = 0; //0是从一个点开始扩散，参考点不变；1是参考点随着扩散会移动  int mergeThreshold = 60; //合并像素差值  bool merge\_small\_block = 0; //是否合并小区域  int lowThreshold = 30; //像素值少于lowThreshold需要被合并  int hiThreshold = 100; //被合并的像素需要合并到大于hiThreshold的地方 |

判断与中心像素点的距离是否在之内

|  |
| --- |
| int calPixel(Vec3b pixelPre,Vec3b pixelNow) {//两个像素点pixelPre与pixelNow之间的距离平方  return pow((pixelNow[0] - pixelPre[0]), 2) + pow((pixelNow[1] - pixelPre[1]), 2) + pow((pixelNow[2] - pixelPre[2]), 2);  }  bool checkColor(Mat& src, int x, int y, int x1, int y1) {  Vec3b pixelPre = src.at<Vec3b>(x, y);  Vec3b pixelNow = src.at<Vec3b>(x1, y1);  return (calPixel(pixelNow, pixelPre) < hs);  } |

**Mean-Shift算法的核心流程**如下：

1、寻找离当前点的距离小于的点，并通过颜色距离过滤掉另一部分点

|  |
| --- |
| for (int i = row - hr; i <= row + hr && i < src.rows; i++) {  if (i < 0)i = 0;  for (int j = col - hr; j <= col + hr && j < src.cols; j++) {  if (j < 0)j = 0;  if (checkLen(src, i, j, row, col) && checkColor(src, i, j, row, col)) { |

2、将上述操作之后保留下来的点的位置向量和记录下来

|  |
| --- |
| xsum += i;  ysum += j;  ++count; |

3、在遍历所有的周围点后，计算该像素点移动的位置

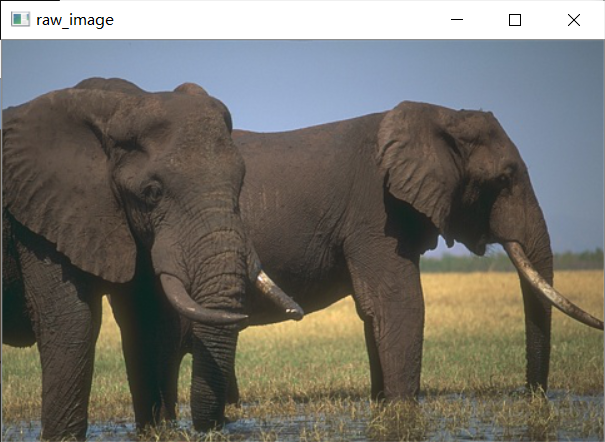
|  |
| --- |
| xsum = xsum / count;  ysum = ysum / count; |

4、判断是否到了迭代终点，即颜色变化小于阈值ColorThreshold或迭代次数超过上限，如果到达终点则停止迭代，否则开启下一轮迭代

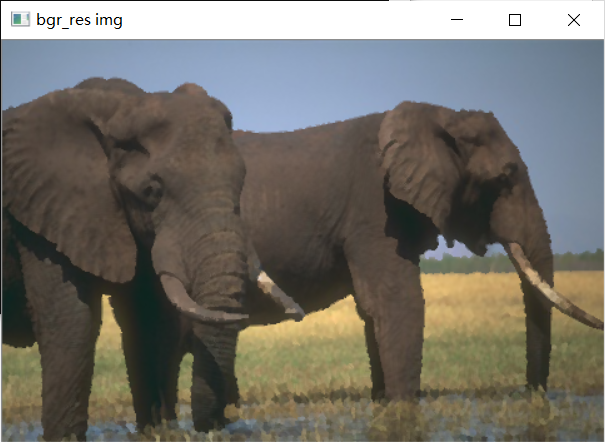
|  |
| --- |
| Vec3b pixelNow = src.at<Vec3b>(xsum, ysum);  if (calPixel(pixelPre, pixelNow) < ColorThreshold || step > iter) {  x = xsum;  y = ysum;  return;  }  else findMean(src, xsum, ysum, step + 1); |

Mean-Shift迭代之后的结果如下：

原图：



Mean-Shift算法后的图：

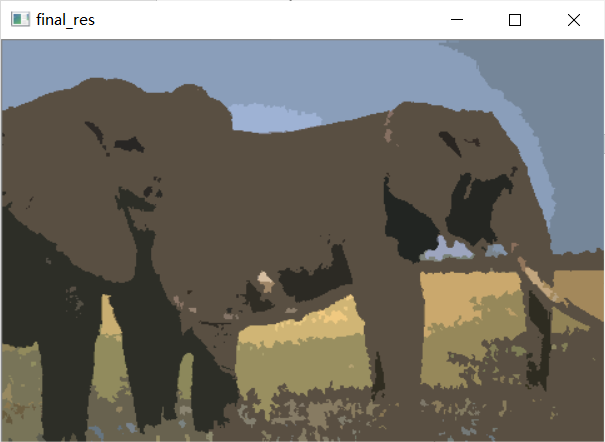


**hr=16,hs=32**

可以看见，每个像素点都有一定的变化，在一定区域内的像素点都漂移到同一个地方，也就是说在一小块区域中像素值都是一样的了，但是这个结果我们并不满意，还需要进行进一步的处理，也就是合并相似区域。具体算法如下：

|  |
| --- |
| Mat dst = Mat::zeros(bgr\_dst.size(), bgr\_dst.type());  for (int row = imageRows - 1; row >= 0; row--) {  for (int col = imageCols - 1; col >= 0; col--) {  //如果当前像素点未被访问过，那么进行bfs来合并周围区域  if (!marked[row][col]) {  //用vec装应当合并的像素点  vec.clear();  //所有合并像素点r,g,b的和  b = 0, g = 0, r = 0;  //当前开始出发合并的像素点  curPixel = bgr\_dst.at<Vec3b>(row, col);  //bfs寻找四邻域相似的像素点  bfs(marked, partImage, row, col, bgr\_dst);  //将寻找到的所有像素点平均的rgb赋给新图像dst  mergeImg(bgr\_dst, dst);  id++;  }  }  } |

像素合并结果



**mergeThreshold = 60**

填充结果：

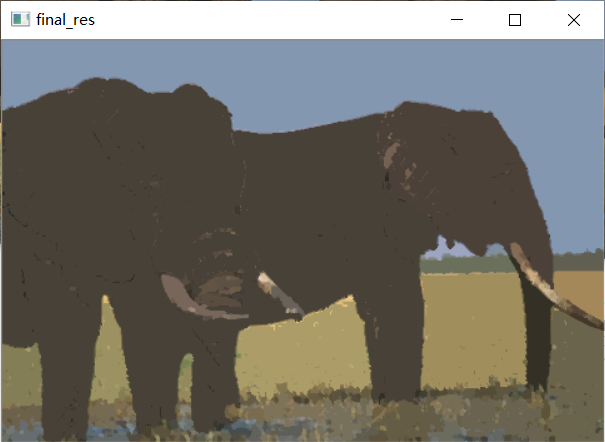


每种颜色都是一块区域。

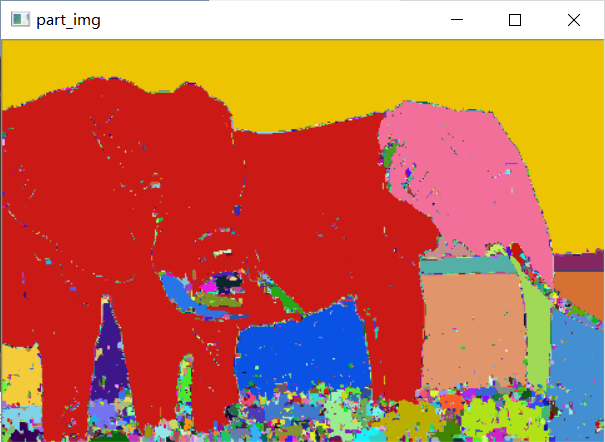
上述的填充结果是从一个像素点开始扩展，在扩展的过程中都是以开始扩展的点作为参照点进行合并，所以可以看到大象上面的天空被分成了三层，结果可能还有点瑕疵，所以，我还扩展了一种合并方案，就是在扩展过程中，变扩展边动态的将参照点变成当前出队列的点，这样就会分的更明显，相同的区域会分为同一块，但是此时合并像素的阈值要调整的小得多，一般在10以内，因为如果是动态调整参照像素，则邻近的两个像素颜色变化是很小的，如果稍微调高了一点，那么整幅图都会被分为同一块，效果十分差劲，经过我多次调参尝试，最佳的参数范围在5~8piexl的距离，这种模式只需要修改一个参数，然后调整合并阈值



下面是该模式的结果图像



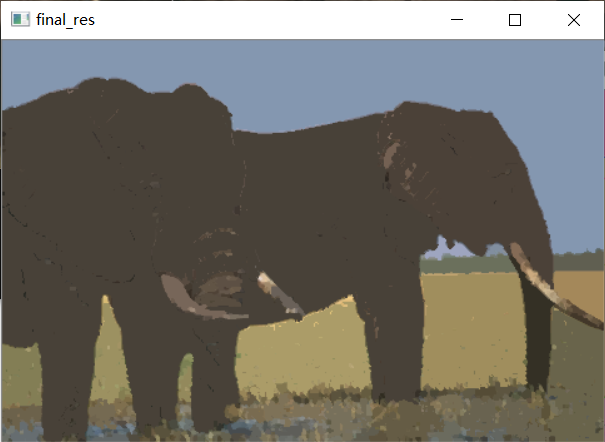
**动态更新参考点结果，mergeThreshold = 8**

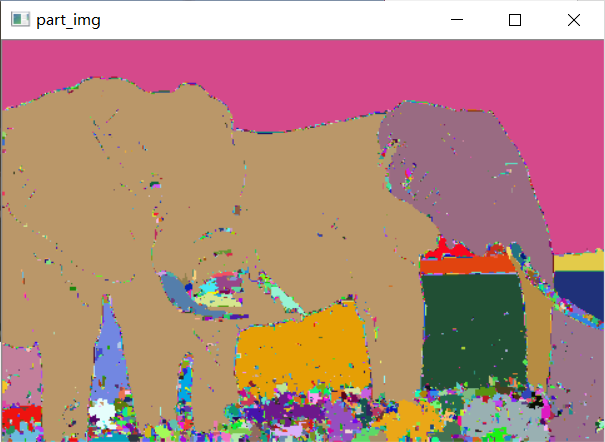


我们会发现，在这种模式下，会产生很多未被合并的小区域图像，使得结果看起来非常糟糕，所以，图像还需要进行进一步的处理，即兼并小区域。兼并小区域的思想就是，在上一步bfs合并相似区域的过程中，记录下了每个区域的像素点个数，然后我们再次通过遍历图像，找到区域中的像素个数少于lowThreshold的区域，然后再次通过bfs寻找到周围的像素个数大于hiThreshold的像素区域，将该区域合并进大区域，就实现了兼并小区域。算法如下

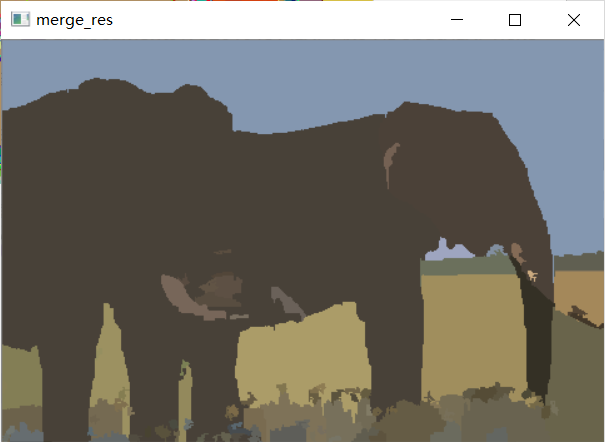
|  |
| --- |
| void mergeSmall(Mat &src,Mat &colorImg) {  for (int row = 0; row < imageRows; row++) {  for (int col = 0; col < imageCols; col++) {  int curID = partID[row][col]; //当前区域的ID编号  int partCount = part\_count[curID]; //找到当前区域的像素个数  if (partCount > lowThreshold)continue; //如果像素个数高于阈值，则不进行合并  search(src,colorImg, row, col); //否则通过搜索进行小区域合并  }  }  } |

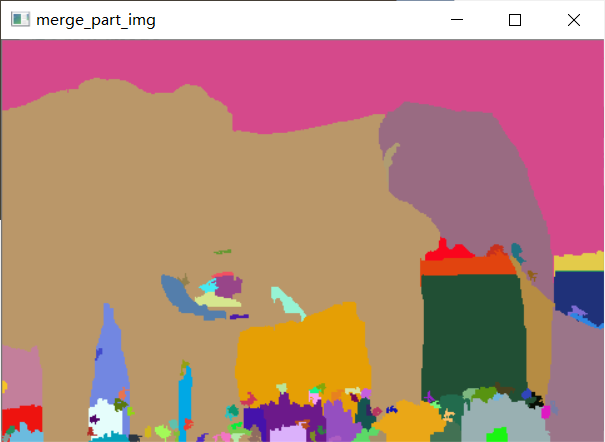
对比结果如下





**合并前，mergeThreshold = 8**



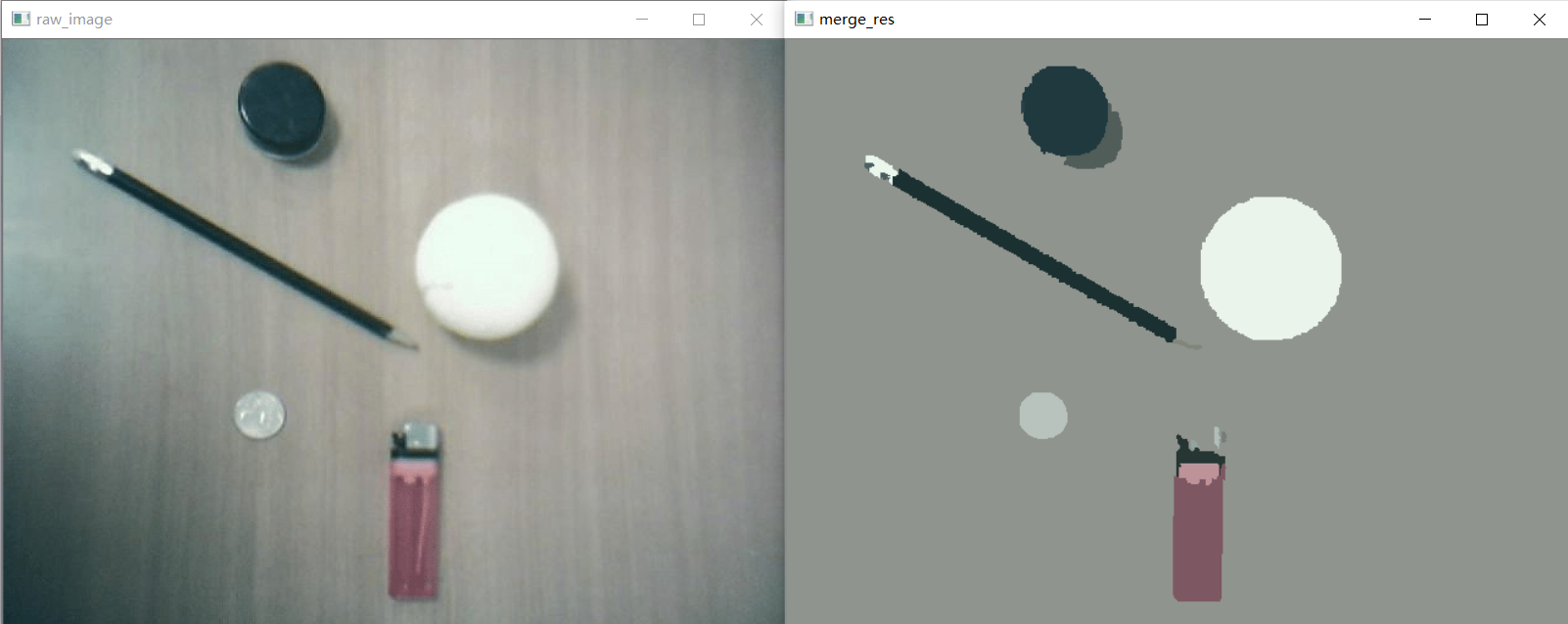


**合并后**

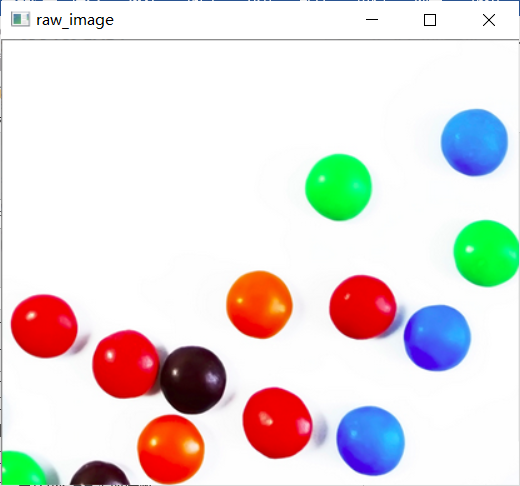
**lowThreshold = 30，hiThreshold = 100**

可以看到，最后的分割结果十分理想。

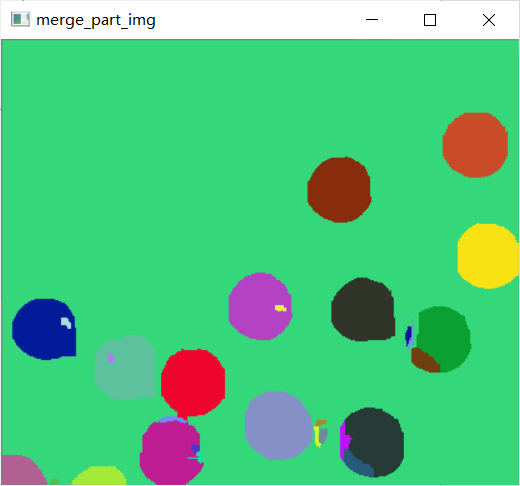
## 七、结果展示





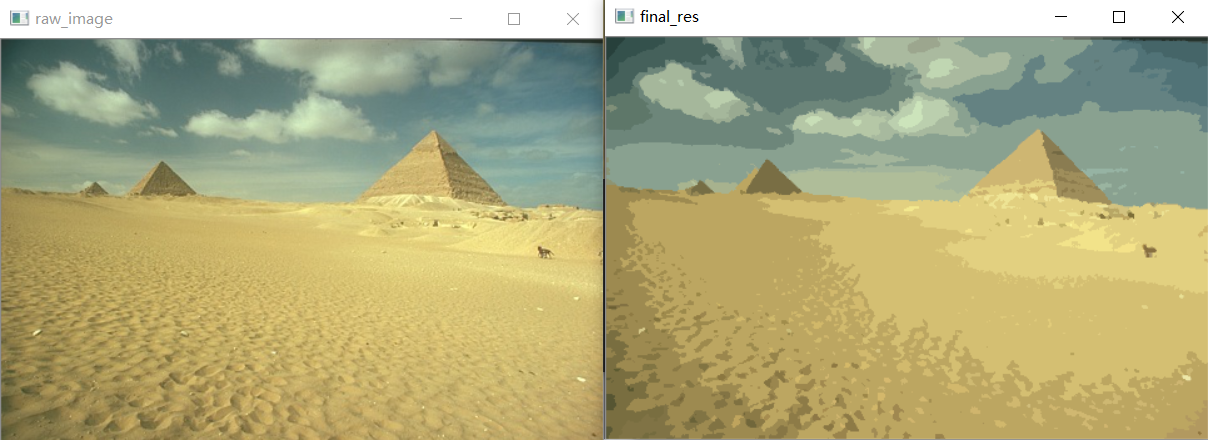




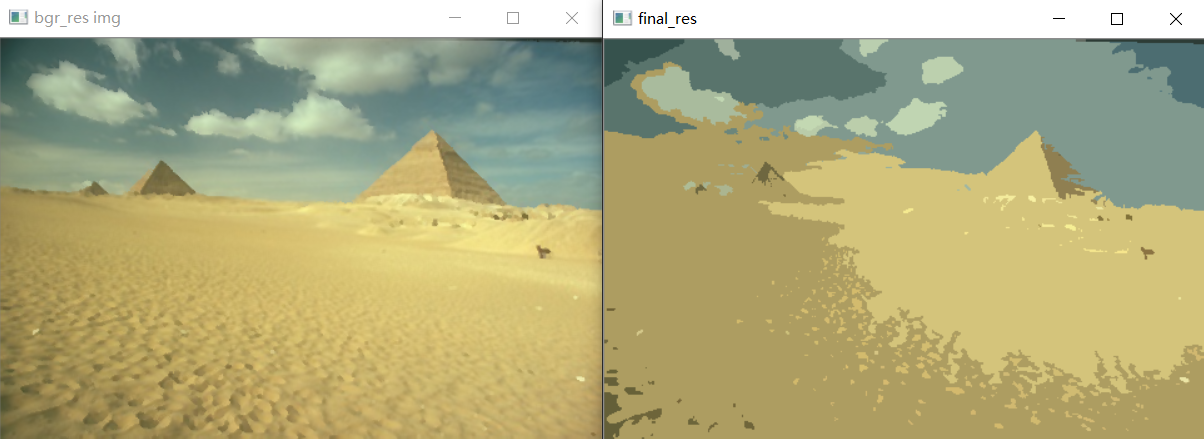


## 八、算法评价

Mean-Shfit算法的时间复杂度较高，需要将每个像素点遍历一次，然还还需要迭代T次，假设图片的大小为，那么其时间复杂度为，在遍历每个点时，还需要进行圆形范围的搜索，我是通过圆的外接正方形来寻找圆形区域所以时间复杂度应修正为,搜索半径r对该算法时间的影响非常关键，当r设为8时，每张图片的平均处理时间约为6秒，当r设为16时，每张图片的平均处理时间达20秒，耗时较多。对合并像素的阈值colorThreshold的设置也很关键，如果想要将图像分的细一些，那么最好设置为20~30，如果想将图像分割的部分少一些，那么需要设置为50~80。



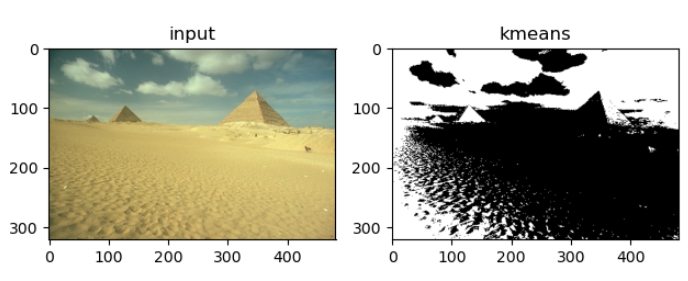
**colorThreshold=30**

****

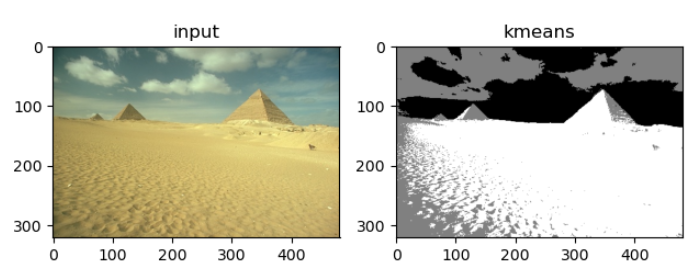
**colorThreshold=65**

与k-means算法比较：

k-means算法是通过聚类的无监督学习来进行图像分割，需要指定聚类的中心点数，该数就是需要进行分割出的图像的结果类数，因为该算法是通过无监督学习，每次聚类中心的位置都是随即指定，所以该算法的结果不可预知，并且调试也不方便



K=2的k-means



K=3的k-means

从上述结果来看，个人感觉基于k-means分割的效果不如mean-shift的好，但是k-means的时间复杂度远比mean-shift要低，不需要一秒就能得到一个图像分割的结果，k-means时间复杂度为，其中k是聚类中心个数, n是样本点数，T是迭代次数。

# 实验二

## 实验目的

实现一种图像匹配算法，并对算法所能达到的效果进行分析。

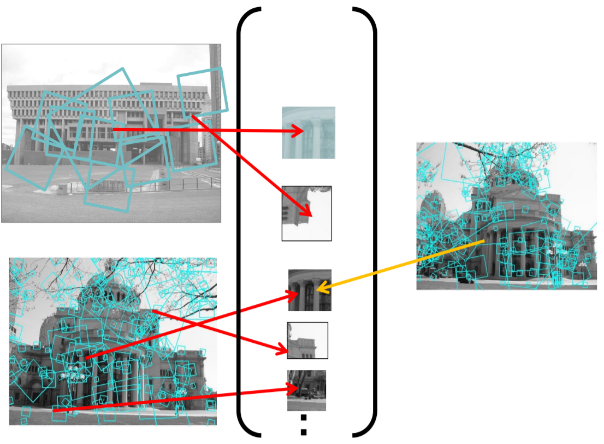
## 实验内容

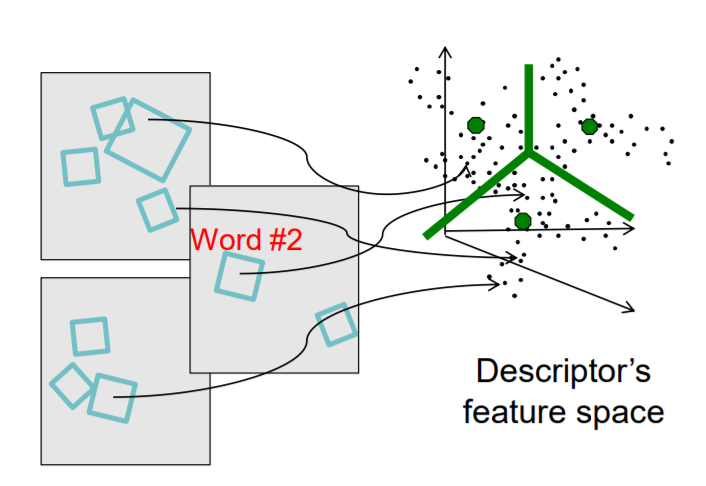
实现了基于sift+BOW算法的图像匹配算法

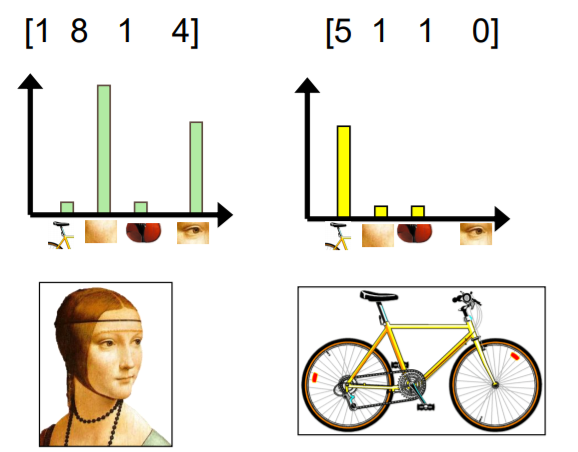
## 三、算法原理

SIFT即尺度不变特征变换，是用于图像处理领域的一种描述。这种描述具有尺度不变性，可在图像中检测出关键点，是一种**局部特征描述子。**所围的尺度不变性，就是说，对于一张图片，不论是将其旋转还是放大缩小，其检测出来的特征都是不变的，不会因为尺度变换而改变。通过这个特性，我们再结合BOW(bag of words)模型，就能实现一个简单的图像匹配算法。

最初的Bag of words，也叫做“词袋”，在信息检索中，Bag of words model假定对于一个文本，忽略其词序和语法，句法，将其仅仅看做是一个词集合，或者说是词的一个组合，文本中每个词的出现都是独立的，不依赖于其他词是否出现，或者说当这篇文章的作者在任意一个位置选择一个词汇都不受前面句子的影响而独立选择的。

现在Computer Vision中的Bag of words来表示图像的特征描述也是很流行的。大体思想是这样的，假设有5类图像，每一类中有10幅图像，这样首先对每一幅图像划分成patch（可以是刚性分割也可以是像SIFT基于关键点检测的），这样，每一个图像就由很多个patch表示，每一个patch用一个特征向量来表示，假设用Sift表示的，一幅图像可能会有成百上千个patch，每一个patch特征向量的维数128，如右图所示。  
      接下来就要进行构建Bag of words模型了，假设Dictionary词典的Size为100，即有100个词。那么咱们可以用K-means算法对所有的patch进行聚类，k=100，我们知道，等k-means收敛时，我们也得到了每一个cluster最后的质心，那么这100个质心（维数128）就是词典里德100 个词了，词典构建完毕。

  
      词典构建完了怎么用呢？是这样的，先初始化一个100个bin的初始值为0的直方图h。每一幅图像不是有很多patch么？我们就再次计算这些patch和和每一个质心的距离，看看每一个patch离哪一个质心最近，那么直方图h中相对应的bin就加1，然后计算完这幅图像所有的 patches之后，就得到了一个bin=100的直方图，然后进行归一化，用这个100维的向量来表示这幅图像。对所有图像计算完成之后，就可以进行分类聚类训练预测之类的了。



然后我们将每张图片的100维向量当作训练集，该向量对应的图片标签为标签，使用SVM进行对特征向量进行训练。最后，对于待预测图片，我们按照上述步骤生成该图片的特征向量，然后使用训练好的分类器进行分类预测，由于SVM分类结果只会输出一类，所以将最高的那个概率的80%作为阈值，只要输出概率高于该阈值，则可以将该图片作为匹配的图片输出。

## 四、算法的Python实现

1、所有图片的SIFT特征点提取，k-means聚类的数据构建

|  |
| --- |
| **def** detect\_SIFT(file\_path, n\_features):  *"""  检测描述特征* **:param** *n\_features: 检测前n\_features个sift特征* **:param** *file\_path:* **:return***: 标签，大矩阵，每包含每张图像的描述的列表  """* feature\_list = [] *# 存储每个图像的描述特征* input\_x = [] *# 用来聚类的大特征* labels = [] *# 存储图像标签* **for** image **in** os.listdir(file\_path):  filename = os.path.join(file\_path, image)  label = image.split(**'.'**)[0]  *# print(filename)* sift = cv2.xfeatures2d.SIFT\_create()  img = cv2.imread(filename)  gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  kpG, desG = sift.detectAndCompute(gray, **None**) *# 获取关键点和特征描述* **if** desG **is None**: *# 判断特征点是否存在* **continue** feature\_list.append(desG) *# 添加一个描述特征* labels.append(label) *# 添加对应的标签* input\_x.extend(desG) *# 合并到大矩阵中  # print(str(len(labels)) + " images detected")* **return** labels, input\_x, feature\_list |

2、训练集的特征向量构建

1）对所有sift特征向量进行聚类，得到聚类中心

|  |
| --- |
| kmeans = KMeans(n\_clusters=n).fit(train\_x) print(**'clusters has created successfully!'**) clusters = kmeans.cluster\_centers\_ |

2）将每个sift点与聚类中心比较，分配到一个聚类中心去，并对该聚类中心对应的特征向量加一

|  |
| --- |
| x = get\_all\_feature(train\_feature, n, clusters)  *# 用一个sift特征描述向量构建一个训练输入向量* **def** get\_features(descriptor, centers, word\_size):  feature\_vec = np.zeros([word\_size])  **for** col **in** descriptor:  diffMat = np.tile(col, (word\_size, 1)) - centers *# 将该列的特征（128\*1）扩大word\_size(聚类中心个数)倍然后与每个中心相减* sqSum = (diffMat \*\* 2).sum(axis=1) *# 平方相加* distances = sqSum \*\* 0.5 *# 开根* index = np.argsort(distances) *# 得到排序后的索引* feature\_vec[index[0]] += 1 *# 最相近的那个中心加一* **return** feature\_vec  *# 创建所有图片的训练输入特征向量* **def** get\_all\_feature(desc\_vec, word\_size, clusters):  vec = np.zeros((len(desc\_vec), word\_size), **'float32'**)  **for** i **in** range(len(desc\_vec)):  vec[i] = get\_features(desc\_vec[i], clusters, word\_size)  **return** vec |

3、对特征向量进行训练

|  |
| --- |
| *# 归一化* sc = StandardScaler() x\_train = sc.fit\_transform(x\_train) x\_test = sc.fit\_transform(x\_test)  *# 训练模型* classifier = SVC(C=20, probability=**True**) classifier.fit(x\_train, y\_train) |

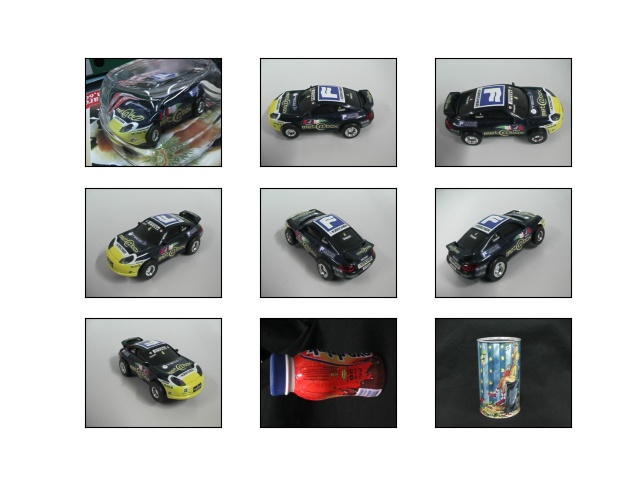
4、预测所有测试集的数据，并将预测结果显示出来，以便对算法的效果进行分析和判断。

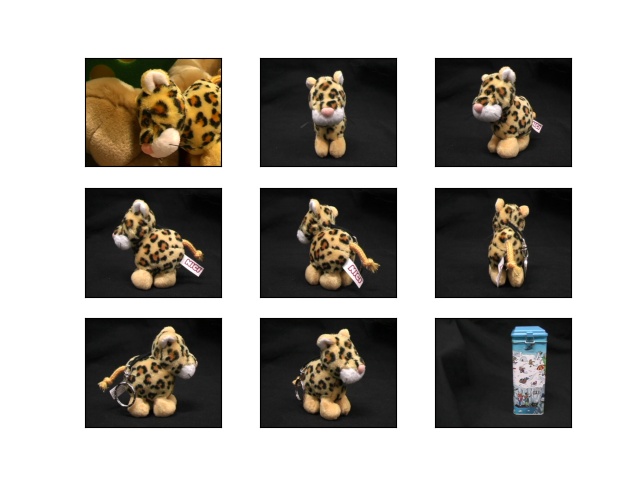
|  |
| --- |
| prob = classifier.decision\_function(x\_test)  res = np.sort(prob) threshold = res[:, -1] \* x **for** i **in** range(threshold.shape[0]):  index = np.argwhere(prob[i, :] > threshold[i]).tolist()  index = [x[0] **for** x **in** index]  img\_list = []  **for** j **in** index:  img\_name = y\_train[j]  path = os.path.join(train\_path, img\_name + **'.tif'**)  img\_list.append(path)  src = os.path.join(test\_path, y\_test[i] + **'.tif'**)  img\_list.insert(0, src)  src = y\_test[i]  save\_res(src, img\_list) |

5、超参数的设置与主函数

|  |
| --- |
| percentage = 0.8 *# 0.8 大于最相似的图像概率\*percentage以上概率的图像* center\_nums = 100 *# 100 聚类中心的数量* create\_train\_and\_test(n\_features) create\_feature(center\_nums) predict2(percentage) |

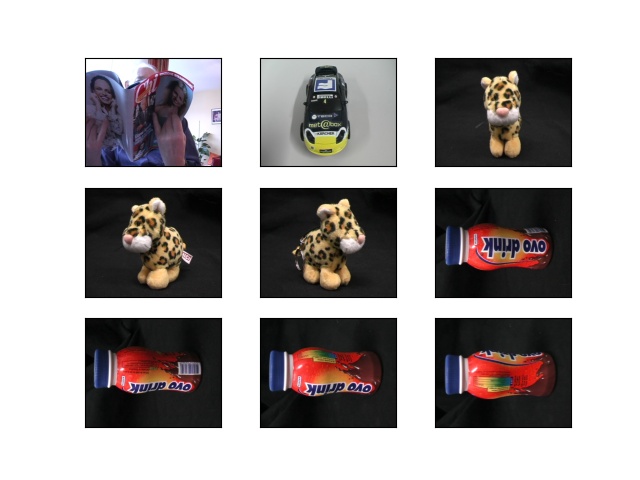
## 五、结果展示





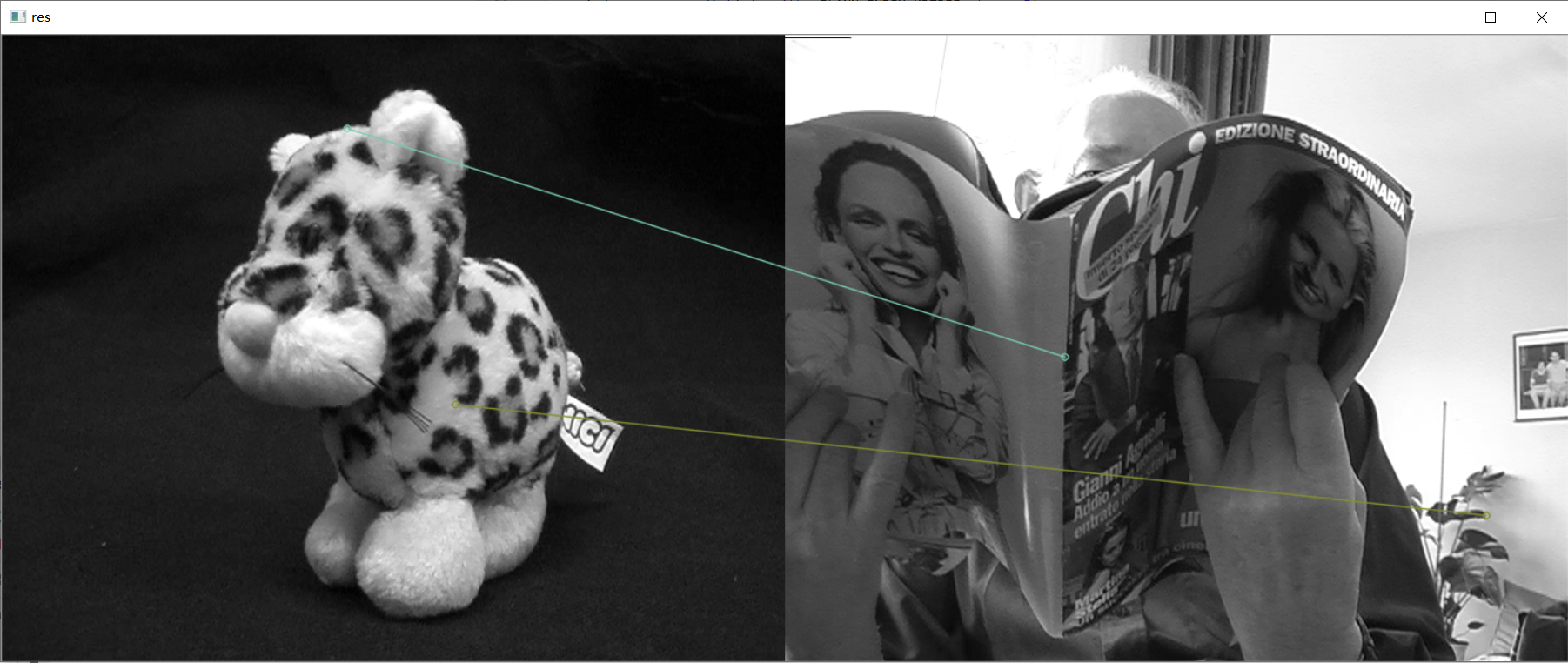
## 六、算法评价



从这张图来看，我们发现没有一个匹配的图，将该图的sift匹配结果一张一张输出来查看具体发生了什么情况：







可以看到，对于该图的匹配，算法找到了匹配的sift点，但是很显然，这几个匹配上的sift点都不正确，因为sift算法匹配的是特征点，并没有任何结构性的特征，如纹理特征等，所以这种匹配模式完全是基于图片的内容进行匹配，对于不同的图片，上面的点特征很可能会相似，就如上面这几张图片所见，虽然人眼看见觉得毫不相关，但是对于电脑来说，他们就是相似的图片。如果加上一些纹理特征进行匹配，那么匹配的精度肯定会更高，因为纹理特征相当于计算机读出了图片中的物体的形状，可以根据物体的形状来判断是否相似，而不是仅仅单纯的通过图像上的点来判断是否相似。

# 实验三

## 实验目的

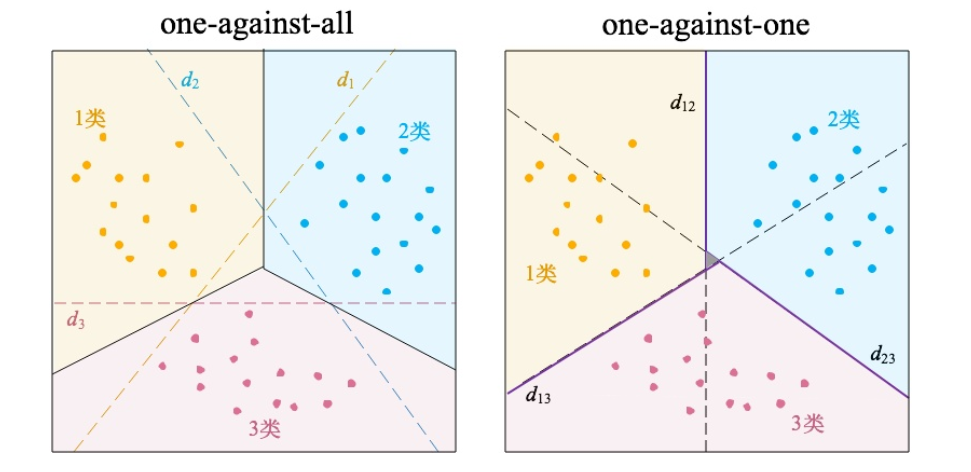
实现一种图像识别算法，并对算法所能达到的效果进行分析。

## 实验内容

基于SVM的手写体数字识别的实现

## 算法原理

支持向量机分类算法最初只用于解决二分类问题，缺乏处理多分类问题的能力。后来随着需求的变化，需要svm处理多分类分为。目前构造多分类支持向量机分类器的方法主要有两类: 一类是“同时考虑所有分类”方法，另一类是组合二分类器解诀多分类问题。



第一类方法主要思想是在优化公式的同时考虑所有的类别数据，J.Weston 和C.Watkins 提出的“K-Class 多分类算法”就属于这一类方法。该算法在经典的SVM理论的基础上，重新构造多类分类型，同时考虑多个类别,然后将问题也转化为-个解决二次规划(Quadratic Programming,简称QP)问题，从而实现多分类。该算法由于涉及到的变量繁多，选取的目标函数复杂，实现起来比较困难，计算复杂度高。

第二类方法的基本思想是通过组合多个二分类器实现对多分类器的构造，常见的构造方法有“一对一”(one-against-one)和“一对其余”(one-against-the rest两种。 其中“一对一”方法需要对n类训练数据两两组合，构建n(n- 1)/2个支持向量机，每个支持向量机训练两种不同类别的数据，最后分类的时候采取“投票”的方式决定分类结果。“一对其余”方法对n分类问题构建n个支持向量机，每个支持向量机负责区分本类数据和非本类数据。该分类器为每个类构造一个支持向量机， 第k个支持向量机在第k类和其余n-1个类之间构造一个超平面，最后结果由输出离分界面距离wx+ b最大的那个支持向量机决定。Sklean里的SVM包使用的就是一对其余分类，训练十个分类器，每个分类器的一边是该数字本身，另一边是其他的九个数字，从而实现了十种手写数字的分类与识别。

## 算法的python实现

1. 读取mnist数据集中的训练集和测试集

|  |
| --- |
| **def** load\_mnist\_data(path, kind):  labels\_path = os.path.join(path, **'%s-labels.idx1-ubyte'** % kind)  images\_path = os.path.join(path, **'%s-images.idx3-ubyte'** % kind)  **with** open(labels\_path, **'rb'**) **as** lbpath:  magic, n = struct.unpack(**'>II'**, lbpath.read(8))  labels = np.fromfile(lbpath, dtype=np.uint8)  **with** open(images\_path, **'rb'**) **as** imgpath:  magic, num, rows, cols = struct.unpack(**'>IIII'**, imgpath.read(16))  images = np.fromfile(imgpath, dtype=np.uint8).reshape(len(labels), 784)  **return** images, labels  train\_images, train\_labels = load\_mnist\_data(path, **'train'**) test\_images, test\_labels = load\_mnist\_data(path, **'t10k'**) |

1. 将测试集和训练集的输入特征向量进行归一化，以便拟合分类器

|  |
| --- |
| X = preprocessing.StandardScaler().fit\_transform(train\_images) XT = preprocessing.StandardScaler().fit\_transform(test\_images) |

1. 取测试集的前10000个数据来进行测试

|  |
| --- |
| XT = XT[0:10000] test\_labels = test\_labels[0:10000] |

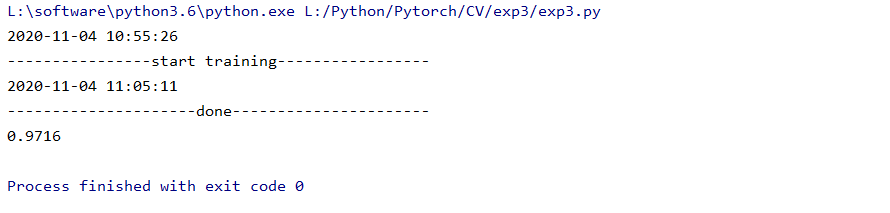
1. 拟合分类器并保存分类器，以便测试使用

|  |
| --- |
| print(time.strftime(**'%Y-%m-%d %H:%M:%S'**)) print(**"----------------start training-----------------"**) clf = SVC(C=100) *# model\_svc = svm.SVC(kernel='poly', C=0.1, gamma=0.01)* clf.fit(X, train\_labels)  **with** open(**'./model.pkl'**, **'wb'**) **as** file:  pickle.dump(clf, file) print(time.strftime(**'%Y-%m-%d %H:%M:%S'**)) print(**"---------------------done----------------------"**) |

1. 用测试集对分类器进行评估，并保存分类错误的图片，以便调试，打印精度

|  |
| --- |
| **with** open(**'./model.pkl'**, **'rb'**) **as** file:  clf = pickle.load(file) predict = clf.predict(XT) precision = np.mean(predict == test\_labels)  **for** i **in** range(predict.shape[0]):  **if** predict[i] != test\_labels[i]:  cv2.imwrite(**"errorimg/"** + str(i) + **'\_label'** + str(test\_labels[i]) + **'\_pred'** + str(predict[i]) + **'.jpg'**,  test\_images[i].reshape(28, 28))  print(precision) |

## 结果展示与评价



六万多张图片训练了十分钟左右，最后的精度达到了97.16%，可以说是相当不错的，我们再来查看一下分类错的图片：

label:9,pred:4，label:7,pred:4，label:2,pred:7， label:9,pred:8

label:3,pred:2，label:4,pred:2，label:2,pred:7，label:5,pred:3

这些是预测错误的一部分结果，大体上来看，可以发现有些数字写的十分歪曲，即便是让人眼来区分都不一定能分得出来，所以可以说明，分类器的分类效果已经很好了，对于一些很歪的图片识别不出来也情有可原。

## 自己手写的字体的识别

原始手写图像：



转换为28\*28的二值图像

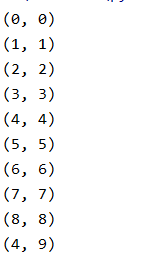
|  |
| --- |
| **for** imgs **in** os.listdir(test\_path):  img\_path = os.path.join(test\_path, imgs)  img = cv2.imread(img\_path)  gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)  resize\_img = cv2.resize(gray, (28, 28))  ret, binary\_img = cv2.threshold(resize\_img, 127, 255, cv2.THRESH\_BINARY\_INV)  cv2.imwrite(**'./binarydata/'** + imgs, binary\_img)  img\_list.append(binary\_img)  label\_list.append(eval(imgs.split(**'.'**)[0])) |

对图像进行预测

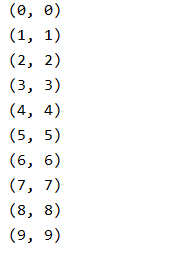
|  |
| --- |
| f = img\_list[0].reshape(1, -1) **for** i **in** range(1, len(img\_list)):  f = np.vstack((f, img\_list[i].reshape(1, -1)))  f = preprocessing.StandardScaler().fit\_transform(f) pred = model.predict(f) pred = list(pred) **for** i **in** range(len(label\_list)):  **if** label\_list[i] != pred[i]:  cv2.imwrite(**"testerrorimg/"** + str(i) + **".jpg"**,img\_list[i]) |

预测结果：



误将9预测成了4，，可能是由于将图片缩小到28\*28之后图像变得不连续导致的，我再画一张粗笔写的9，再次进行预测

，二值图像为，最后看到预测结果

，全部都预测正确。