

地球科学与环境工程学院

**《数字图像处理》**

实 验 报 告

**实验名称： 监督分类及精度评定**

**指导老师： 叶沅鑫**

**班 级： 遥感一班**

**姓 名： 史祖浩**

**学 号： 2021114677**

**2023年 12 月 17 日**

目录

[实验——监督分类（最小距离法、最大似然法） 3](#_Toc154159169)

[实验目的与任务 3](#_Toc154159170)

[实验原理 3](#_Toc154159171)

[最小距离法 3](#_Toc154159172)

[最大似然法 4](#_Toc154159173)

[编程思路 4](#_Toc154159174)

[最小距离法 4](#_Toc154159175)

[最大似然法 4](#_Toc154159176)

[设计实现 5](#_Toc154159177)

[最小距离法分类 5](#_Toc154159178)

[最大似然法分类 6](#_Toc154159179)

[精度评价 10](#_Toc154159180)

[运行结果 16](#_Toc154159181)

[最小距离法分类结果及精度评定： 16](#_Toc154159182)

[最大似然法分类结果及其精度评定： 18](#_Toc154159183)

[实验总结 19](#_Toc154159184)

# 实验——监督分类（最小距离法、最大似然法）

## 实验目的与任务

将1000\*500的RGB影像及其分类标签数据（标签数据中有五类样本）

数据说明：

文件夹中有4个数据train.tiff，train\_label.tiff， test.tiff， test\_label.tiff

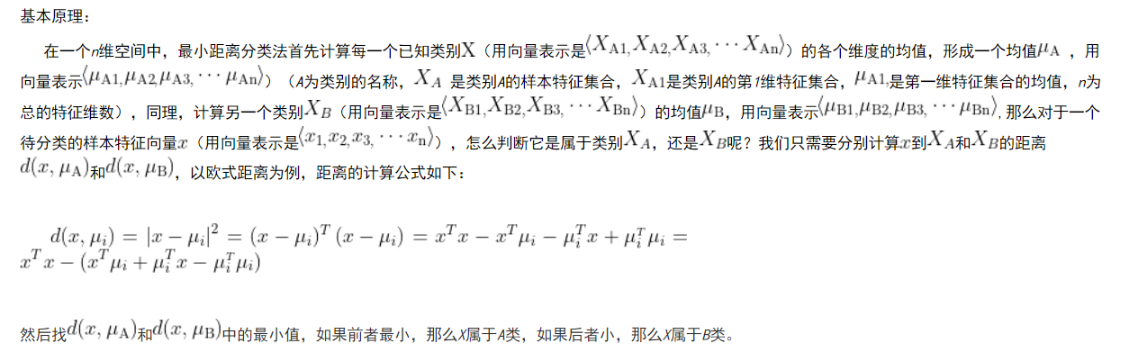
1. train. tiff 为用于训练的样本数据
2. train\_label.tiff 为用于训练的样本标签数据
3. test.tiff 为用于分类的数据
4. test\_label.tiff 分类的标准结果，用来评价算法的分类精度

* 利用标签给的五类数据（0，50，100，150，200），采用最小距离法和最大似然法对test.tiff 数据进行分类。
* test影像为待分类影像，分为5类，分类结果同样（0，50，100，150，200）表示。
* 将分类结果与test\_label进行比较，并进行分类精度分析。

## 实验原理

### 最小距离法

**最小距离分类法**是分类器里面最基本的一种分类方法，它是通过求出未知类别向量X到事先已知的各类别（如A，B，C等等）中心向量的距离D，然后将待分类的向量X归结为这些距离中最小的那一类的分类方法。



### 最大似然法

基本原理：

假设有两个事件A，B，我们通过先验的知识，知道A发生的条件下，x也发生的概率是P(x|A)； B发生的条件下，x也发生的概率是P(X|B)，那么，现在有一个事件x发生了，我们能否判断这个事件x是在A条件下，还是在B条件下发生的可能性大些呢？也就是要求出P（A|x）和P（B|x）哪一个最大？对分类问题而言，哪一个概率大，我们就说x属于哪一类。

它是以建立统计识别函数为理论基础，依据典型样本训练方法进行分类的技术。即根据已知训练区提供的样本，通过选择特征参数，求出特征参数作为决策规则，建立判别函数以对各待分类影像进行的图像分类的一种方法。

最大似然法假设每类地物所对应的波段数据都呈正态(高斯)分布，因此每个地物类别实际上由对应的正态分布的期望和方差决定。

## 编程思路

### 最小距离法

1. 获取类别均值向量means
   * 利用所有样本,计算每个类别的均值像素（特征中心）
2. 分类函数minDistClassify
   * 输入:均值向量,测试像素
   * 将测试像素转为向量testVec
   * 遍历每个类别i
     + 计算测试样本与类i中心的欧式距离dist
     + 如果距离更近,更新最小距离minDist和类别标号predLabel
   * 返回最终最小距离的类别predLabel
3. 精度评估
   * 获取 Accuracy 和 Kappa系数
   * 输出混淆矩阵等指标

### 最大似然法

1. 获取类别模型参数
   * 先验概率probbly

统计每个类别的样本数，计算比例，获得各类别的先验概率P(Yi)

* + 高斯分布参数:均值向量means,协方差矩阵covMat
    - 平均向量means

利用Eigen求取每类别样本的均值向量

* + - 协方差矩阵covMat
      1. 计算类别内样本与均值的差矩阵
      2. 计算差矩阵的转置与自身的乘积得到协方差矩阵

1. 分类函数 maxLikelihoodClassify
   * 输入:先验概率,高斯参数,测试像素
   * 将测试像素转为向量
   * 遍历每个类别i
     + 计算高斯分布密度概率
     + 与先验概率相乘得到后验概率
   * 返回后验概率最大的类别
2. 精度评估
   * 计算 Accuracy 和 Kappa系数
   * 输出混淆矩阵等指标

## 设计实现

### 最小距离法分类

获取均值向量（我本次实验将计算均值向量、协方差矩阵、先验概率等放在一个函数中实现，因此在最大似然法中详细介绍）

**分类函数**minDistClassify实现：

1. 传入均值向量，测试像素
2. 循环遍历每一个类别i，计算样本与其中心的欧氏距离
3. 更新最小距离和类别编号

|  |
| --- |
| // 最小距离分类  int minDistClassify(vector<Vector3d> means, int trainLabels[], int numTrain, Pixel testPix)  {  Vector3d testVec = Pix2Vec(testPix);  double minDist = INT\_MAX;  unsigned int predLabel;  for (int i = 0; i < numTrain; ++i)  {  double dist = caldistance(testVec, means[i]);  if (dist < minDist)  {  minDist = dist;  predLabel = trainLabels[i];  }  }  // 预测类别  return predLabel;  } |

计算欧式距离：

|  |
| --- |
| // 计算两个向量距离  double caldistance(Vector3d a, Vector3d b)  {  return sqrt(pow(a[0] - b[0], 2) + pow(a[1] - b[1], 2) + pow(a[2] - b[2], 2));  } |

像素结构体转为向量：

|  |
| --- |
| // 像素到向量的转换  Vector3d Pix2Vec(Pixel a)  {  VectorXd V(3);  V << a.r, a.g, a.b;  return V;  } |

### 最大似然法分类

获取类别模型参数（先验概率、高斯分布参数（均值向量、协方差矩阵））

|  |
| --- |
| //获取均值、协方差矩阵、先验概率  tuple<vector<Vector3d>, vector<Matrix3d>, vector<double> > getMeansCovPro(vector<Pixel> train, vector<Pixel> Label)  {  int k0 = 0; int k1 = 0; int k2 = 0; int k3 = 0; int k4 = 0;  MatrixXd ccvv0(Height \* Width, 3);  MatrixXd ccvv1(Height \* Width, 3);  MatrixXd ccvv2(Height \* Width, 3);  MatrixXd ccvv3(Height \* Width, 3);  MatrixXd ccvv4(Height \* Width, 3);  for (int i = 0; i < Width \* Height; i++)  {  if (Label[i].r == 0)  {  ccvv0(k0, 0) = train[i].r;  ccvv0(k0, 1) = train[i].g;  ccvv0(k0, 2) = train[i].b;  k0++;  }  else if (Label[i].r == 50)  {  ccvv1(k1, 0) = train[i].r;  ccvv1(k1, 1) = train[i].g;  ccvv1(k1, 2) = train[i].b;  k1++;  }  else if (Label[i].r == 100)  {  ccvv2(k2, 0) = train[i].r;  ccvv2(k2, 1) = train[i].g;  ccvv2(k2, 2) = train[i].b;  k2++;  }  else if (Label[i].r == 150)  {  ccvv3(k3, 0) = train[i].r;  ccvv3(k3, 1) = train[i].g;  ccvv3(k3, 2) = train[i].b;  k3++;  }  else  {  ccvv4(k4, 0) = train[i].r;  ccvv4(k4, 1) = train[i].g;  ccvv4(k4, 2) = train[i].b;  k4++;  }  }  ccvv0.conservativeResize(k0, 3);  ccvv1.conservativeResize(k1, 3);  ccvv2.conservativeResize(k2, 3);  ccvv3.conservativeResize(k3, 3);  ccvv4.conservativeResize(k4, 3);  // 先验概率  vector<double> probability(5);  probability[0] = (double)k0 / train.size();  probability[1] = (double)k1 / train.size();  probability[2] = (double)k2 / train.size();  probability[3] = (double)k3 / train.size();  probability[4] = (double)k4 / train.size();  // 以向量存储均值，用于最大似然法  vector<Vector3d> means;  MatrixXd a0 = ccvv0.colwise().mean();  a0.resize(a0.rows(), a0.cols());  Vector3d m0 = a0.transpose();  MatrixXd a1 = ccvv1.colwise().mean();  a1.resize(a1.rows(), a1.cols());  Vector3d m1 = a1.transpose();  MatrixXd a2 = ccvv2.colwise().mean();  a2.resize(a2.rows(), a2.cols());  Vector3d m2 = a2.transpose();  MatrixXd a3 = ccvv3.colwise().mean();  a3.resize(a3.rows(), a3.cols());  Vector3d m3 = a3.transpose();  MatrixXd a4 = ccvv4.colwise().mean();  a4.resize(a4.rows(), a4.cols());  Vector3d m4 = a4.transpose();  means.push\_back(m0);  means.push\_back(m1);  means.push\_back(m2);  means.push\_back(m3);  means.push\_back(m4);  MatrixXd cov\_0(ccvv0.rows(), 3);  MatrixXd cov\_1(ccvv1.rows(), 3);  MatrixXd cov\_2(ccvv2.rows(), 3);  MatrixXd cov\_3(ccvv3.rows(), 3);  MatrixXd cov\_4(ccvv4.rows(), 3);  //2 每个样本减去均值  for (int i = 0; i < ccvv0.rows(); i++)  {  cov\_0.row(i) = ccvv0.row(i) - a0;  }  for (int i = 0; i < ccvv1.rows(); i++)  {  cov\_1.row(i) = ccvv1.row(i) - a1;  }  for (int i = 0; i < ccvv2.rows(); i++)  {  cov\_2.row(i) = ccvv2.row(i) - a2;  }  for (int i = 0; i < ccvv3.rows(); i++)  {  cov\_3.row(i) = ccvv3.row(i) - a3;  }  for (int i = 0; i < ccvv4.rows(); i++)  {  cov\_4.row(i) = ccvv4.row(i) - a4;  }  //3 计算  cov\_0 = cov\_0.transpose() \* cov\_0 / (cov\_0.rows() - 1);  cov\_1 = cov\_1.transpose() \* cov\_1 / (cov\_1.rows() - 1);  cov\_2 = cov\_2.transpose() \* cov\_2 / (cov\_2.rows() - 1);  cov\_3 = cov\_3.transpose() \* cov\_3 / (cov\_3.rows() - 1);  cov\_4 = cov\_4.transpose() \* cov\_4 / (cov\_4.rows() - 1);  vector<Matrix3d> Cov(5);  Cov[0] = cov\_0;  Cov[1] = cov\_1;  Cov[2] = cov\_2;  Cov[3] = cov\_3;  Cov[4] = cov\_4;  // vector从0到4分别对应0、50、100、150、200  return make\_tuple(means, Cov, probability);  } |

多元高斯概率密度计算函数：

|  |
| --- |
| // 多元高斯概率密度计算  double MulGaussProbability(VectorXd mean, MatrixXd cov, VectorXd test)  {  int k = mean.size();  double determinant = cov.determinant();  double normalization = 1.0 / (pow(2 \* PI, k / 2) \* sqrt(determinant));  MatrixXd inv\_cov = cov.inverse();  double exponent = -0.5 \* (test - mean).transpose() \* inv\_cov \* (test - mean);  return normalization \* exp(exponent);  } |

分类函数maxLikelihoodClassify：

1. 输入:先验概率,高斯参数,测试像素
2. 将测试像素转为向量
3. 遍历每个类别i
   1. 计算高斯分布密度概率
   2. 与先验概率相乘得到后验概率
4. 返回后验概率最大的类别

|  |
| --- |
| // 最大似然法分类  int maxLikelihoodClassify(vector<double> probbly, vector<Vector3d> means, vector<Matrix3d> cov, int trainLabels[], int numTrain, Pixel testPix)  {  Vector3d tePix = Pix2Vec(testPix);  int maxId = -1;  double maxProb = 0;  for (int i = 0; i < numTrain; i++)  {  double prob = MulGaussProbability(means[i], cov[i], tePix) \* probbly[i];  if (prob > maxProb)  {  maxProb = prob;  maxId = trainLabels[i];  }  }  return maxId;  } |

### 精度评价

根据混淆矩阵计算Accuracy和Kappa系数：

|  |
| --- |
| // 根据混淆矩阵计算Accuracy及Kappa系数  tuple<double, double> getAccuracyKappa(MatrixXi Mat)  {  double sumCount = Mat.sum();  VectorXi rowCount = Mat.rowwise().sum();  VectorXi colCount = Mat.colwise().sum();  // TPi(真正例):对于类i,被正确分类为类i的样本数量。即混淆矩阵对角线元素的值。  int TP0 = Mat(0, 0); int TP1 = Mat(1, 1); int TP2 = Mat(2, 2); int TP3 = Mat(3, 3); int TP4 = Mat(4, 4);  double temp1 = double(TP0 + TP1 + TP2 + TP3 + TP4);  double accuracy = temp1 / Mat.sum();  double P0 = accuracy;  // 计算Kappa系数  // 计算观察一致性P0(已有)：其实就是Accuracy  // 计算期望一致性Pe(已有)：对每行每列求和，再除以总数的平方  double Pe = 0;  for (int i = 0; i < Mat.cols(); i++)  {  double temp1 = rowCount[i] \* colCount[i];  Pe += temp1;  }  Pe = Pe / pow(Mat.sum(), 2);  double kappa = (P0 - Pe) / (1 - Pe);  return make\_tuple(accuracy, kappa);  } |

精度评定:

1. EvaAccuracy函数

- 输入:参考标注像素reference,分类结果result

2. 主要步骤:

- 初始化混淆矩阵Nxx,大小为类别数\*类别数

- 遍历每个像素

- 根据reference判断实际类别i

- 根据result判断预测类别j

- Nxx中第(i,j)元素计数加1

- 计算评价指标:

- 调用getAccuracyKappa

- 计算准确率Accuracy

- 计算Kappa系数

3. 存储与输出评价结果

- 输出混淆矩阵

- 输出Accuracy、Kappa指标

|  |
| --- |
| //精度评定  void EvaAccuracy(vector<Pixel> reference, int\* result)  {  MatrixXi Nxx = MatrixXi::Zero(5, 5);  // 计算各项  for (int i = 0; i < Width \* Height; i++)  {  if (reference[i].r==0)  {  if (result[i]==0)  {  Nxx(0, 0) += 1;  }  else if (result[i]==50)  {  Nxx(0, 1) += 1;  }  else if (result[i] == 100)  {  Nxx(0, 2) += 1;  }  else if (result[i] == 150)  {  Nxx(0, 3) += 1;  }  else  {  Nxx(0, 4) += 1;  }  }  else if (reference[i].r==50)  {  if (result[i] == 0)  {  Nxx(1, 0) += 1;  }  else if (result[i] == 50)  {  Nxx(1, 1) += 1;  }  else if (result[i] == 100)  {  Nxx(1, 2) += 1;  }  else if (result[i] == 150)  {  Nxx(1, 3) += 1;  }  else  {  Nxx(1, 4) += 1;  }  }  else if (reference[i].r == 100)  {  if (result[i] == 0)  {  Nxx(2, 0) += 1;  }  else if (result[i] == 50)  {  Nxx(2, 1) += 1;  }  else if (result[i] == 100)  {  Nxx(2, 2) += 1;  }  else if (result[i] == 150)  {  Nxx(2, 3) += 1;  }  else  {  Nxx(2, 4) += 1;  }  }  else if (reference[i].r == 150)  {  if (result[i] == 0)  {  Nxx(3, 0) += 1;  }  else if (result[i] == 50)  {  Nxx(3, 1) += 1;  }  else if (result[i] == 100)  {  Nxx(3, 2) += 1;  }  else if (result[i] == 150)  {  Nxx(3, 3) += 1;  }  else  {  Nxx(3, 4) += 1;  }  }  else  {  if (result[i] == 0)  {  Nxx(4, 0) += 1;  }  else if (result[i] == 50)  {  Nxx(4, 1) += 1;  }  else if (result[i] == 100)  {  Nxx(4, 2) += 1;  }  else if (result[i] == 150)  {  Nxx(4, 3) += 1;  }  else  {  Nxx(4, 4) += 1;  }  }  }  // 定义接受变量  double Accuracy, Kappa;  tie(Accuracy, Kappa) = getAccuracyKappa(Nxx);  std::cout << "精度评定完成，结果保存于data目录下" << std::endl;  std::fstream f;  f.open(".\\data\\EvaResult.txt", std::ios::out);  //写入的内容  f << "混淆矩阵如下：" << '\n';  for (int i = 0; i < Nxx.rows(); i++)  {  for (int j = 0; j < Nxx.cols(); j++)  {  f << setw(6) << Nxx(i, j)<<" ";  if (j == Nxx.cols()-1)  {  f << '\n';  }  }  }  f << "总体精度Accuracy：" << Accuracy << '\n'  << "Kappa系数：" << Kappa;  } |

## 运行结果

### 最小距离法分类结果及精度评定：



其混淆矩阵及精度评定：



### 最大似然法分类结果及其精度评定：



其精度评定：

文本

描述已自动生成

## 实验总结

由于这是本学期最后一次实验，前面已经积累了相关的GDAL库、Eigen库的相关知识，因此使用这些库来完成一些工作是相当便利高效的。

我的实验总结可分为下面四个部分：

1. GDAL图像数据读取

由于已经熟悉了GDAL的图像数据格式和存储方式，因此在读取图像时，考虑到其有三个波段，因此选择设计了一个Pixel结构体（里面有三个double类型的变量r，g，b），因此使用GDAL读取文件后每一个像素即为定义的一个结构体，并使用C++内置容器vector用于储存每一个结构体；

因此设计了读取图像函数，通过传入设计好的Data类，返回值为存储了包含三波段数据的结构体的vector容器；

然后分别读取train及trainLabel，获取各自的数据vector容器

1. 高斯分类模型参数计算

最初使用循环逐像素判别类别统计概率，发现这种方式效率低下，并且存在大量冗余代码，因此改为利用Eigen的向量化运算，从而可以简化计算过程。不过因为在设计读取图像时，将图像数据存到了结构体中，因此又设计了一个结构体转为向量的函数，从而实现转化，大大地简便了运算过程。

我设计的高斯分类模型参数计算的函数为：传入存储了train和trainLabel图像数据的vector，根据训练标签图像分别计算每一类样本数量和样本数据（rgb），从而完成各类别先验概率、各类别各波段的均值，各类别的协方差矩阵的计算，并返回这些值。

当然在计算高斯分布密度时，也出现了很多问题，例如Eigen库使用时初始化或者重置大小时遇到的析构问题等等（一次忽略了析构问题导致矩阵内部数据全为0，从而计算出错，分类只有一类）；

还有对于协方差矩阵的计算，一开始也是没有直接采用Eigen库的矩阵计算而且计算公式也不正确，导致了计算出错或者计算效率太低，后面也是采用了Eigen库大大简化了运算以及保证了效率和正确率。

1. 高斯分类概率模型计算

一开始，我认为此次最大似然分类应该是独立的多元高斯概率分布模型，因此只需要将各波段高斯概率相乘并乘上相应类别先验概率即可，得到的结果没那么理想，后面通过看PPT、请教老师和与同学交流才确定为多元高斯概率分布（不独立），根据多元高斯概率分布计算公式代入各类别及待分类像素计算概率并获取最大概率及其类别。

1. 混淆矩阵及Kappa系数计算

有了分类后图像及标签图像，计算其混淆矩阵，利用Eigen库还是很简便计算的，多分类的Kappa系数及精确率Accuracy也比较容易得出，最后输出保存至txt文档即可，比较简单。