武汉大学

遥感信息工程学院

《高性能地理计算》课程实习作业

学号:

2021302051202

姓名:

石铠瑞

course name:

基于 pyspark 的 kmean 算法的遥感图像聚类分析

2023 年 11 月 24 日

目录

1. 实习要求

本次实习选题为《基于 pyspark 的 kmean 算法的遥感图像聚类分析》, 旨在通过 spark 用 python 语言编写 kmean 算法对遥感影像做基于像素的分类。其实现步骤主要为以下几步:

1. 基于 Windows 的 pyspark 环境配置
2. 设计遥感图像处理函数
3. 通过 spark 运用 rdd 处理形式编写 kmean 函数与主函数
4. 设定合理参数处理影像并分析聚类效果

2. Spark 环境配置

2.1 安装 jdk

安装 jdk 首先并配置环境变量，在命令行输入 $java - version$ 即可验证安装是否成功。

```
java -version
```

```
C:\Users\scarick>java -version
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

图 2.1　命令行安装成功反馈

2.2 安装 Hadoop 与 Spark

下载 Hadoop 与 Spark 包并写入环境变量。

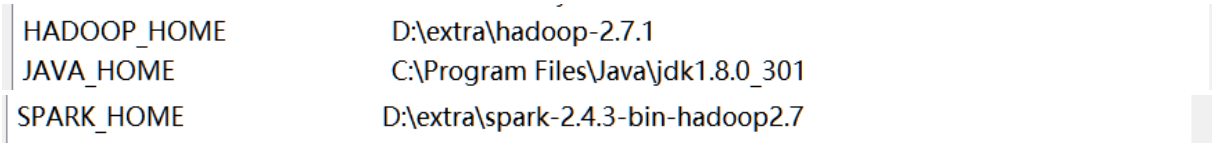


图 2.2　jdk、Hadoop、Spark 环境变量配置

2.3 anaconda 创建虚拟环境

创建 python3.6 的虚拟环境以适配该版本的 Spark 并激活:

```
conda create --name pyspark python==3.6
conda activate pyspark
```

随后，将 Spark 包移至虚拟环境包路径下，安装 py4j 包后即可验证 Spark 安装配置情况，如图所示，即为配置成功。

图 2.3　Spark 安装配置成功

3. 算法实现

算法实现主要由三部分组成：图像处理、Kmean 函数、主函数。

3.1 图像处理

图像处理主要将导入的遥感影像进行处理转化为可处理的数据集。本文编写 Extract_pic.py 文件实现主函数进行图像处理的操作。

首先通过 PIL 包导入 Image 类：

```
from PIL import Image
```

由于导入的遥感影像例图为 RGB 三通道彩色正射影像图，因此先需要单通道灰度化，采用例图如下 (原图为.tif 图像 pdf 展示为 png 格式略有不同)：

图 3.4　遥感正射影像例图

通过函数 tif_rgb2gray() 将.tif 彩色遥感影像转为灰度图像。

```
def tif_rgb2gray():
    rgb_image = Image.open('output.tif')
    image = gray_image = rgb_image.convert('L') #L即为灰度格式
    gray_image.save('output_gray.tif')
    rgb_image.close()
    return image
```

由上述代码可知，输出灰度图像并保存，示例遥感影像如下：

图 3.5　转为灰度图像

在转为单通道灰度图像之后，就可以提取每个像元的像素值作为 spark 进行 kmean 计算的数据集。

```
def get_pixelvalue(image):
    pixels = list(image.getdata()) #list列表读取存储像素值
    with open('pixel_values.txt', 'w') as file:
        for pixel in pixels:
            # 写入像素值到文本文件
            file.write(f'{pixel}\n')
```

通过函数 get_pixelvalue(image)，将灰度图像每个像素值读入到一个 list 中，并写入文本文件，供 spark 以 rdd 形式读取，文本文件如下：

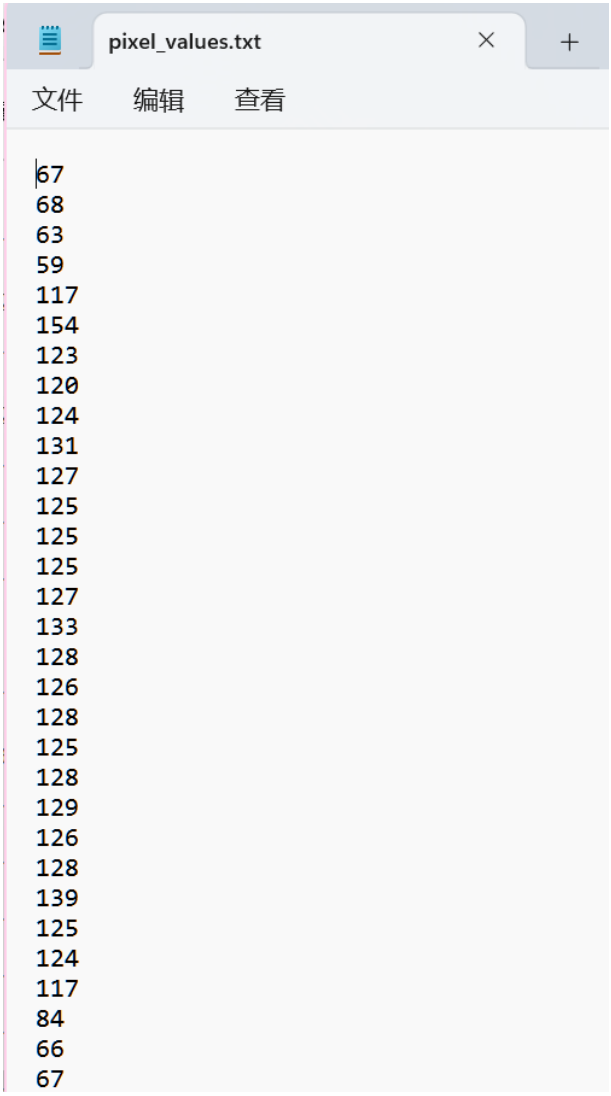


图 3.6　图像像素值文本文件

如图，每一个像素值以行形式保存至文本文件中。

最后，当 Kmean 算法训练完成时，会输出如数据集一样的分类后的像素值列表，需要再由该数组/列表输出图像：

```
def produce_img(pixels,width,height):
    new_image = Image.new("L", (width, height))
    new_image.putdata(pixels)
    new_image.save('classify.png')
    return new_image
```

通过 produce_img 函数读入像素值序列 pixels 以及输出图像高度宽度，将 pixels 里面的元素值拟合输出分类后图像’classify.png’。

3.2 Kmean 函数编写

通过 spark，利用 rdd 数据形式编写 Kmean 函数，本文通过编写函数 Kmean.py 实现。

1) 首先，导入类与函数：

```
import numpy as np
from pyspark.sql import Row
from pyspark.ml.linalg import Vectors
```

2) 其次，定义 Kmean 函数，将读入的 rdd 数据集导入，设定聚类个数预设值 k=3，迭代次数预设值 epoch=10 次。并且，由于首次迭代需要随机生成聚类中心坐标，因此设定随机种子 seed 确保每次调用函数时取到的随机值不一样。

```
def kmeans(data_rdd, k = 3, seed = None, epoch = 10):
```

3) 进入函数，首先应当初始化类中心：

```
  # 初始化聚类中心
    np.random.seed(seed)
    length = 1
    lb = 0
    up = 256
    centroids = np.random.randint(lb, up, size=(k, length))
```

调用随机种子，确保随机序列可控，确定聚类中心维度 length 的大小，由于本次实习是针对遥感图像的聚类分析，因此读取的数据集为像元像素值，即为一维数据，所以聚类中心坐标维度 length=1。

由于不论分几类，所有像素值都在 0-255 之间，因此可设定随机数上下限 lb=0,up=256，因此通过 np.random.randint 函数生成整数随机数时，聚类中心的坐标值一定为 0-255 的整数值，符合实际情况。最后，定义生成聚类中心的数组大小 size，由聚类个数 k 的大小与数据维度共同决定。

4) 进入迭代，当迭代次数小于 epoch 时不断迭代：

```
i = 0
   while i < epoch:
       print('i = %d' % i)
```

5) 由于不断更新迭代的聚类中心，包括初次随机生成的聚类中心可能存在聚类中心顺序错误，因此通过函数将其重新排列。

```
        # 需要将中心点按照大小排列
        centroids =
            np.sort(centroids.flatten()).reshape(centroids.shape)
```

该函数中，首先通过 centroids.flatten() 函数将中心点展平成一维，因为由上一步可知中心点数组 centroids 的大小 size 为 (k,length)，而在本次实习中 length=1，因此 centroids 为 k 行一列的一个列向量，因此在排序时需要先将其展平。

之后通过 sort() 函数排序，再 reshape 为原来的排列形状。

6)计算每个样本点到每个聚类中心的距离

该步骤乃 Kmean 算法核心步骤，代码如下:

```
        distances = data_rdd.map(lambda point:
            (np.argmin([np.linalg.norm(point - c) for c in
            centroids]), point))
```

首先通过 np.linalg.norm(point - c) for c in centroids，从 centroids 中读取中心点，用每个特征值与其求差值，然后用 norm 函数求每个向量的长度，即每个点与各个中心点的欧式距离，通过 argmin 这个函数找到最小值的索引。例如: 假设聚类中心个数 k=3，遍历计算一个点与三个中心点的欧式距离，若与类 2 距离最近，即返回 centroids 类中心索引—— “2”。

将上述返回的索引值 (即为 Index)，与数据点的 point 值组成元组 (Index,point)，通过 rdd 数据类型的 map 函数 data_rdd.map(lambda point: (Index,point)) 将 data_rdd 以行形式读取每个点的值 point 再构成元组后以行形式的方式返回给 distances 这个新的 rdd 数据类型数据。因此，可以得到每个点的分类结果。

7) 分类完成后执行 i++，如若 i 已经达到预设分类次数则没有必要更新中心点再进行分类，跳出循环即可。

```
        i += 1
        if i < epoch:
            clusters = distances.groupByKey()

            # 计算新的聚类中心
            new_centroids_rdd = clusters.map(lambda cluster:
```

```
                np.round(np.mean(np.array(list(cluster[1])), axis=0)))
            new_centroids = new_centroids_rdd.collect()

            # 更新聚类中心
            centroids = np.array(new_centroids)
```

由上述代码可知，先通过 clusters = distances.groupByKey() 中的 groupByKey 函数对分好类别的数据集进行排序，这是 spark 中自带的函数可以直接调用并对 rdd 类型数据集使用。该函数会对 rdd 数据集第一个列的变量分类排序，在 distances (cluster_index, point) 中第一列为该点所属的类别，因此 clusters 将通过 groupByKey 函数把同一类的点归类。得到的 clusters 为 rdd 数据格式，并且有多少类就有多少行，每行第一个元素代表类号，第二个元素是一个迭代器代表该类别中的所有点，即 ($group_i$，迭代器迭代读取存储所有点)。

得到 clusters 后需要通过该次分类情况计算新的类中心。同样，通过 rdd 数据格式中的 map 函数读取 clusters 中每一类中所有点，并用 np.array(list(cluster[1])) 并转换成数组格式。再用 np.mean(... , axis=0)) 计算每一类数据点各自的均值，且均值即为新的类中心，最后存入 centroids 更新中心点。

8) 在达到预设训练次数后跳出循环，此时分类已完成。

```
    # 预测
    predictions = distances.map(lambda x:
        Row(features=Vectors.dense(x[1]), prediction=int(x[0])))

    return predictions
```

同样，用 map 函数逐行读取 distances 中数据，在上文中已表明 distances 的组成形式为 (Index,point)，此时我们仅需要创建名为 predictions 的 rdd 数据格式，将 distances 中的 point 放入 predictions 中的第一列特征值'features' 中，以代表各个点的数据，将 Index 放入第二列预测值'prediction' 中，以代表该点的分类结果。

```
def kmeans(data_rdd, k = 3, seed = None, epoch = 10):
    ...
    return predictions
```

最后，返回 prediction。这样整个 Kmean 函数就完成了，通过读入 rdd 数据集、预设聚类个数 k、训练次数 epoch，就能输出基于 Kmean 算法分类 rdd 数据集 prediction。

3.3 主函数编写

编写程序 imageprocess.py，导入所需函数

```
import findspark
findspark.init()

import numpy as np
from pyspark import SparkContext
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import when,col, lit,
    monotonically_increasing_id as mi
from pyspark.ml.clustering import KMeans
from pyspark.ml.linalg import Vectors
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from pyspark.ml.evaluation import ClusteringEvaluator
from Extract_pic import bmp2tif, tif_rgb2gray, get_pixelvalue,
    produce_img
from PIL import Image
# 引入自己编写的kmean
from Kmean import kmeans
```

3.3.1 数据集准备

读入图片进行处理：

```
    # process image
    filepath = 'input.bmp'
    bmp2tif(filepath)
    image = tif_rgb2gray()
    get_pixelvalue(image)
    height = image.height
    width = image.width
    image.close()
```

如上述，在读取图像 image 后，存入图片的宽高便于最后分类图像合成。

```
# build RDD
    spark = SparkContext(appName="KMeans_pyspark",
        master='local[8]') # SparkContext
    SparkSession(spark)
```

之后创建 spark 分布式数据处理服务器集群，由于没有两台电脑，因此使用本地机器上的 8 个线程（local[8]）作为 Spark 集群的主节点。创建 spark 服务集群可以用 sparksession 直接创建，也可以先用 sparkcontext 创建再 sparksession 化，由于版本问题担心 sparksession 部分函数不可用，因此本文采用 sparkcontext 创建 spark 再 sparksession 化，这样就算出现 sparksession 用不了的函数也能够用 sparkcontext 中的函数处理。

随后，需要读入数据集。但再次之前，由于需要将数据集 (像元灰度值) 转变为模型训练的特征向量，因此编写函数 f(x) 如下：

```
    # convert data to feature vector
    def f(x):
        rel = {}
        rel['pixel_values'] = Vectors.dense(float(x[0]))
        return rel

    # read data
    data_rdd = spark.textFile('./pixel_values.txt').map(lambda
        line:line.split('\t')).map(lambda p: Row(**f(p)))
```

读取数据集为 rdd 数据格式，调用 spark 中的 textFile 函数，通过 map 的方式，将每一行以缩进分隔符的方式读取数据，并将该行读取的数据集用 f(x) 中的 Vectors.dense 函数转化为特征向量。尽管本次实习中数据集一行只有一个数据，但用缩进的方式读取每一行数据可以应对高维数据。最后将读入的数据集保存至 data_rdd。

3.3.2 kmean 分类

```
    classify_rdd = kmeans(data_rdd, k=3)
```

调用 kmeans 函数，导入 rdd 数据集，设定分类个数 k=3，迭代沿用默认值 epoch=10，将分类结果传入 classify_rdd。

3.3.3 聚类赋值

由于我现在得到的 classify_rdd 中有两列，第一列为原始像素值，第二列为分类结果。要想将分类结果可视化需要将分类结果转化为像素值。

```
    kmeans_results = classify_rdd.toDF()
    kmeans_results = kmeans_results.withColumn("new_pixel_value",
        lit(0))
```

因此这里先将处理完的 rdd 转为 dataframe 格式，通过 spark 中处理 dataframe 数据的函数 withColumn() 添加一个新列”new_pixel_value”，并用 lit(0) 添加一个常量列，赋值为 0。

为了处理根据类别给对应类别的像元赋上对应像素值，通过聚类数 k 将 0-255 分为 k-1 段，这样算上 0 与 255 就得到 k 种均分的像素值。

```
    new_pixel_value_list = []
    new_pixel_value_list.append(0.0)
    for i in range(1, k-1):
        new_pixel_value_list.append(i*(256.0/(k-1)))
    new_pixel_value_list.append(255.0)
```

首先创建存储新像元的列表，并将像素值 0 填入第一个，再将 256 分为 k-1 份，依次填入列表共 k-2 次，最后再将 255 填入。因为此处只是为了制作分类像素集，因此不存在像素值除不尽如何化整问题，最后合成图像时会自动处理，只需要分出每一类点即可。

```
    for cluster in range(0, k):
        new_pv = new_pixel_value_list[cluster]
        kmeans_results = kmeans_results.withColumn("new_pixel_value",
            when(col("prediction") == cluster,
            new_pv).otherwise(col("new_pixel_value")))
```

最后遍历 k 个类别，根据不同类别赋予新像素值 new_pv，利用 withcolumn 函数将分类结果 prediction 等于对应类别的数据找出来，将它的 new_pixel_value 列填入新像素值 new_pv，其他数据则沿用之前的 new_pixel_value。遍历所有类别之后每一个点都得到了一个它类别对应的新像素值。

3.3.4 分类可视化

通过 dataframe 数据格式的函数 collect 收集得到的新像素值。

```
    pixels_data = kmeans_results.select("new_pixel_value").collect()
    pixels = [row.new_pixel_value for row in pixels_data]
```

通过列表推导式遍历 pixels_data 中的每一行，然后从每行中提取 new_pixel_value 的值，最终形成一个包含这些值的列表 pixels。

```
    produce_img(pixels, width, height)
    new_image = Image.open('classify.png')
    new_image.show()
    new_image.close()
```

最后，将这些像素值列表 pixels 与图像长宽赋予给之前编写的图像生成函数即大功告成，生成分类图像 classify.png。

4. 成果展示

4.1 聚类可视化

当聚类个数 k=3 时，分类结果如下图：

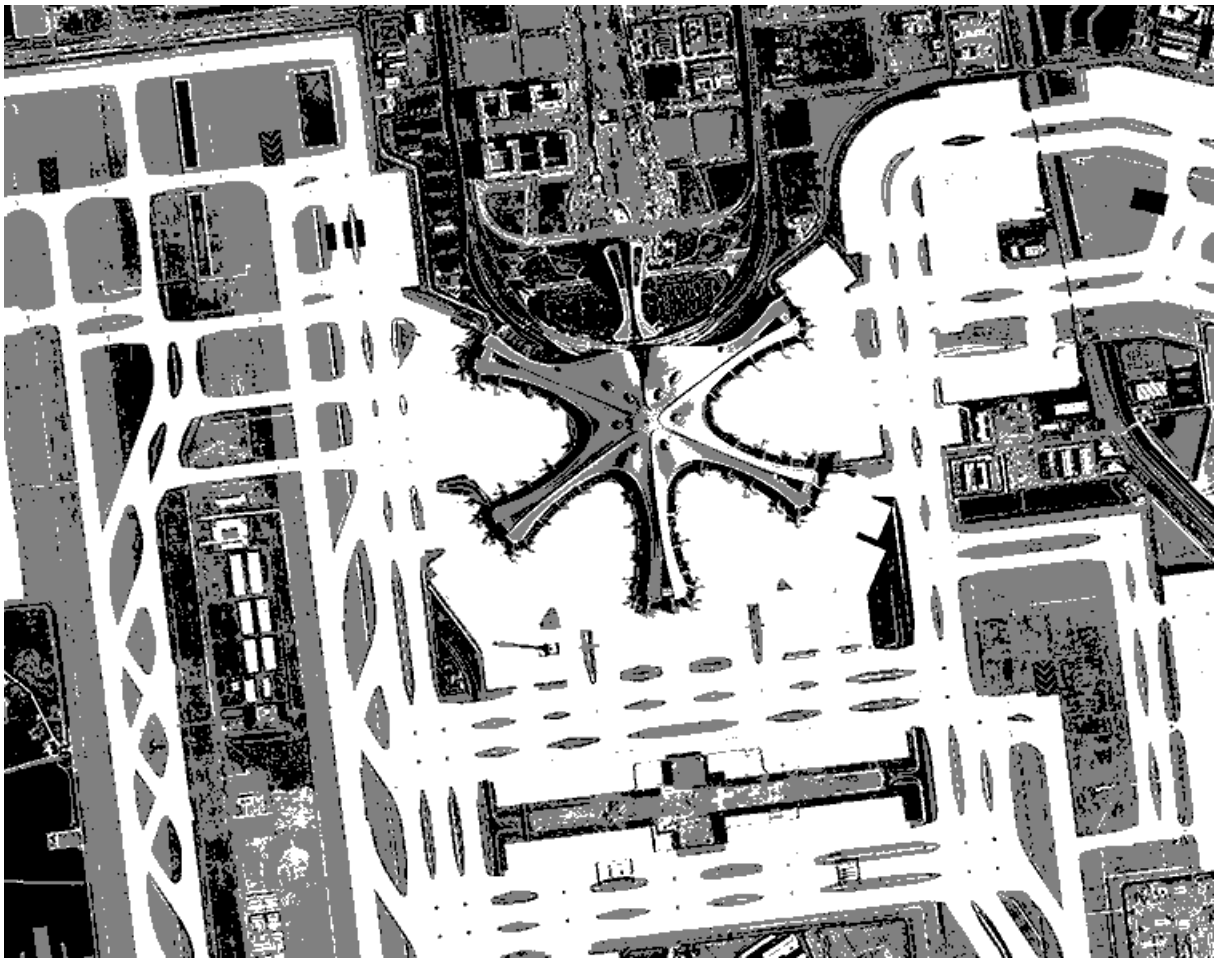
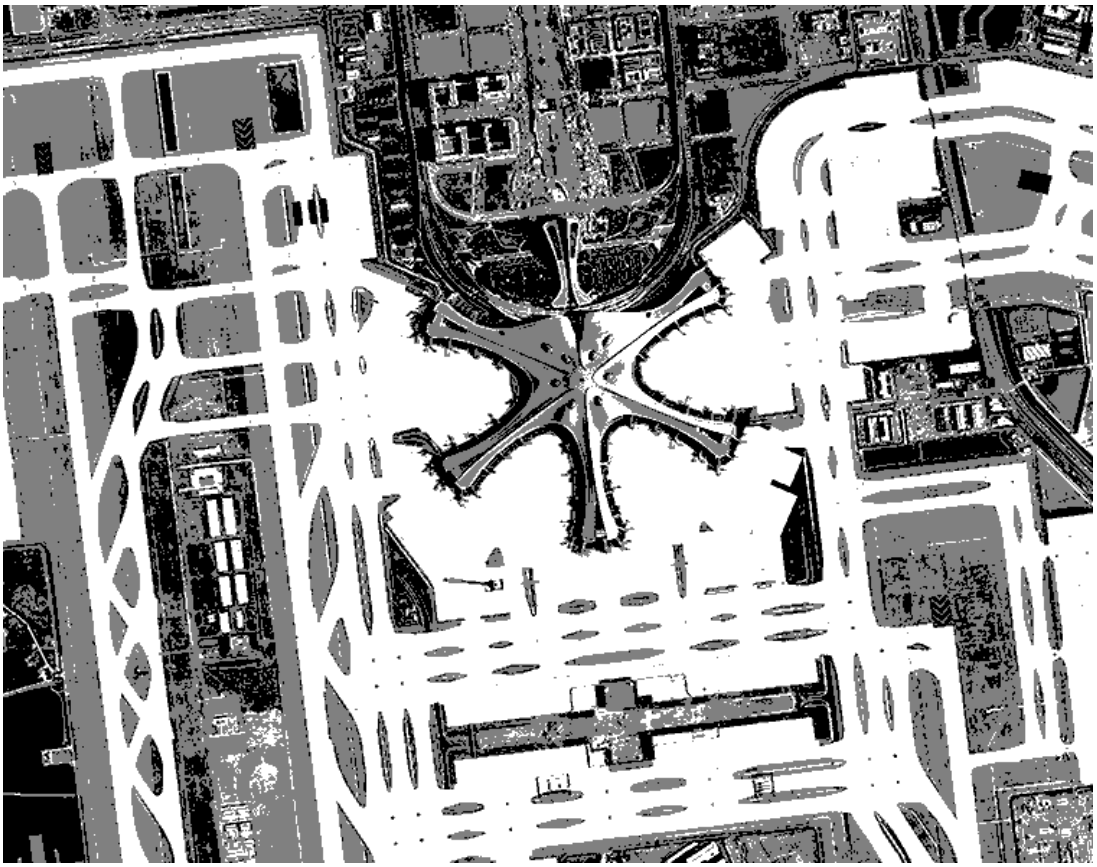


图 4.7　k=3-epoch=10

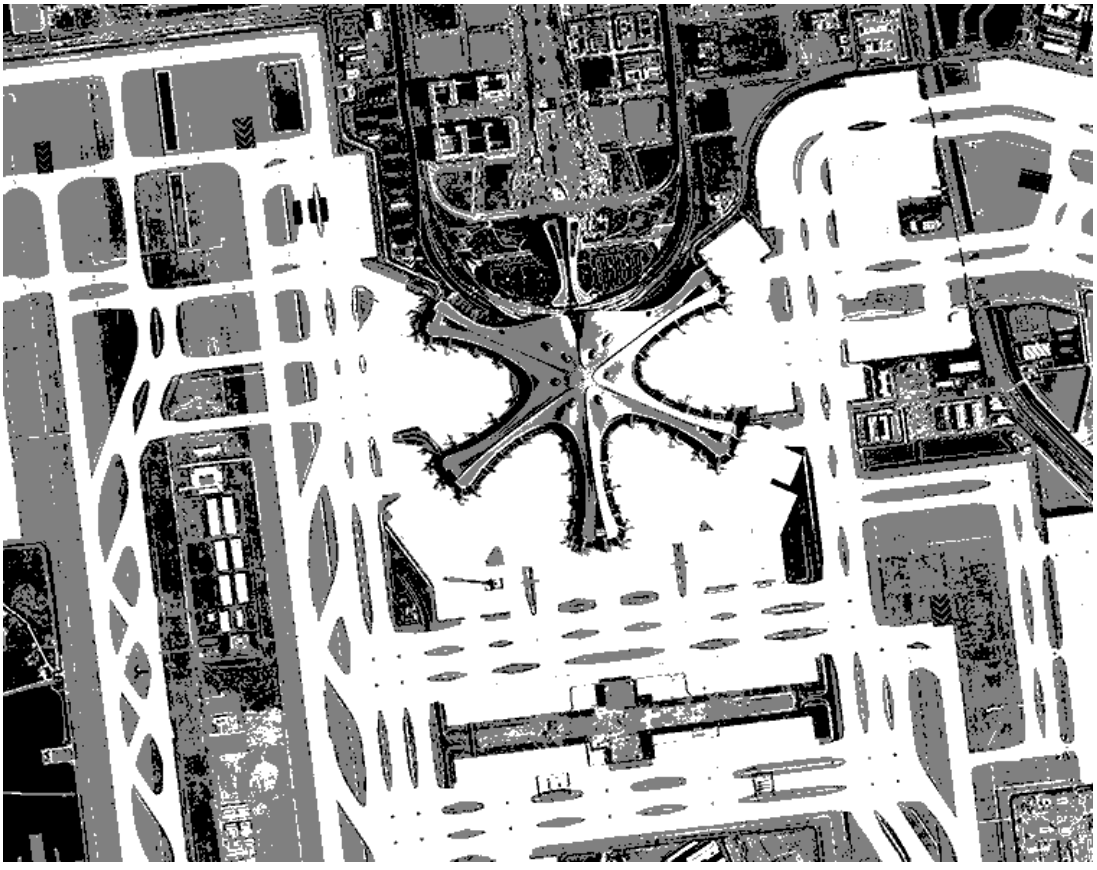
可以看出，道路，草坪，建筑物都十分清晰，甚至一些地物细部都能显示出来，例如图像左上角飞机跑道方向指示牌处：

图 4.8　细部分类效果

但是，仅仅从单个模型的观感上觉得还可以是不足以判别模型的好坏。因此，本文还用 pyspark 自带的 MLlib 机器学习模型库中的 Kmean 来进行对比，下面是在 k=3 时，两个模型的分类可视化结果：



(a) mykmean



(b) mllibkmean

图 4.9　不同模型对比图

可以看出，两次分类目视结果非常相近，因此输出模型的聚类中心分别为：

	cluster1	cluster2	cluster3
mykmean	89	137	163
MLlibkmean	65.9	109.6	158

可以看出虽然从视觉层面上几乎无法分别聚类效果，但是直接从聚类效果上来看二者的聚类中心明显不同，因此本文选取分类评价指标进行进一步模型评估。

4.2 Silhouette 轮廓指数模型质量评估

由于无法通过目视判读判断自己写的 Kmean 模型与 mllib 自带 kmean 模型的差距，因此采用 Silhouette 轮廓指数来评估模型质量。

轮廓系数（Silhouette Coefficient）可以用于评估聚类质量，因为它同时考虑了簇内的紧密度和簇间的分离度。对于每个数据点，轮廓系数计算了它与同一簇中其他点的相似度（簇内相似度）和与最近的其他簇中的所有点的相似度（簇间相似度），然后将这两者之差除以两者中较大的值。

轮廓系数的计算公式如下：

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

式中：

- $S(i)$ 是数据点 i 的轮廓系数。
- $a(i)$ 是数据点 i 与同一簇中其他点的平均距离（簇内相似度）。
- $b(i)$ 是数据点 i 与最近的其他簇中的所有点的平均距离（簇间相似度）。

并且，轮廓系数的取值范围在 -1 到 1 之间：

- 如果 $S(i)$ 接近 1，表示数据点 i 被正确地分配到了簇中，簇内相似度高，簇外相似度较低。
- 如果 $S(i)$ 接近 -1，表示数据点 i 更适合被分配到其他簇，簇外相似度较高。
- 如果 $S(i)$ 接近 0，表示数据点 i 在簇的边界上。

因此，聚类的平均轮廓系数越接近 1，表示聚类效果越好。设计函数计算如下：

```
1
2     evaluator = ClusteringEvaluator()
3     # 设置评估指标为'silhouette'
4     silhouette = evaluator.evaluate(kmeans_results)
5     print("Silhouette Score:", silhouette)
```

首先，导入 from pyspark.ml.evaluation import ClusteringEvaluator 类，用于评估聚类模型的性能。通过 evaluator = ClusteringEvaluator() 创建一个 ClusteringEvaluator 的实例，用于评估聚类模型的质量。再使用 ClusteringEvaluator 对 KMeans 模型的结果 kmeans_results 进行评估，计算轮廓系数。

	Silhouette
mykmean	0.8007347
MLlibkmean	0.7994294

可以看到，自创建的模型的 Silhouette 指数更接近 1，表明分类情况更加精准。因此，通过与泛用性较强的 MLlibkmean 相比较，我的模型精度是没有问题的，只是效率不及前者。

4.3 成果展示

在模型精度检验之后，将聚类类别个数提高到 k=5, 迭代次数 epoch=10，带入模型运算，结果如图：

图 4.10　真实图像

图 4.11　k=5-epoch=10 分类图

可以看到，分类效果非常的好。与真实地物对比，面积较大的区域 (像元数量大且灰度值灰度接近) 例如草坪、道路、荒地、建筑等等区域，都得到了很好的分类，地面纹理结构与细部图案也得到了很好的保留，例如箭头和道理纹理。

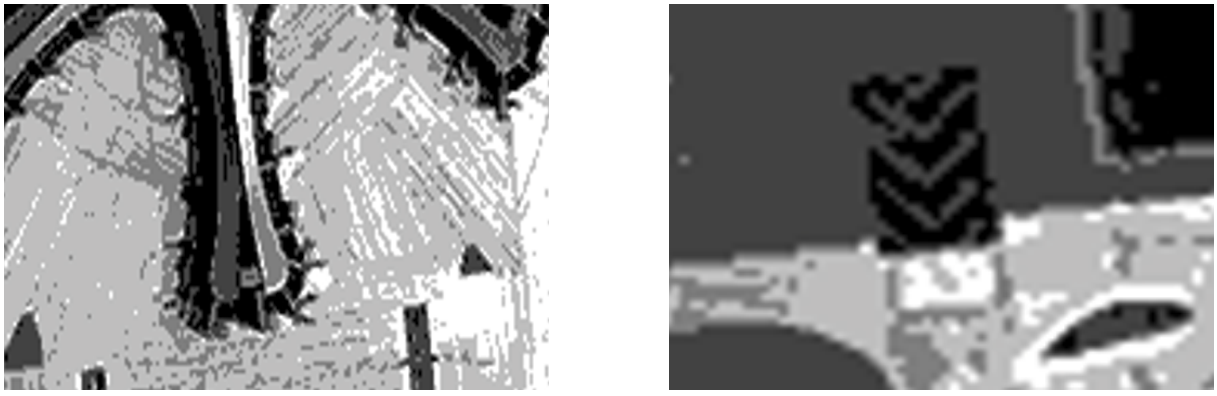


图 4.12　细部分类效果

5. 模型推广与评价

1. 创建 pyspark 目前多用 SparkSession，但我直接用 SparkSession 会导致报错，因此我只能退而求其次先使用传统的 Sparkcontext 创建，再用 SparkSession 激活。时间有限，我未能找到解决方案，可能是版本问题。但是我这样做的好处是 Sparkcontext 与 SparkSession 的相关函数我都能调用，在编写代码时也方便很多。
2. 虽然模型精度较高，但运行速度不及 pyspark 自带算法，可能是有庞大的函数库作支撑。
3. 尽管是使用并行处理算法，运行速度大幅提升，但在处理大尺寸遥感影像时，一般会先采取降采样再进行图像处理，但本文为了检验算法将整个图像录入，一张影像就高达五十万个像素点。
4. 若想要突出分类结果，可进行假彩色图像合成。
5. 调用 MLlib 无需再大幅更改整个程序，只需要更改部分主程序即可。

附 录

本次实习一共编写了四个程序，图像处理板块、Kmean 算法函数、主函数与调用 MLlib 中 kmean 的主函数。

A 图像处理板块

```python
from PIL import Image

def bmp2tif(filepath):
    image = Image.open(filepath).convert('RGB')
    image.save('output.tif', format='TIFF', compression='tiff_lzw')
    image.close()
    return True

def tif_rgb2gray():
    rgb_image = Image.open('output.tif')
    image = gray_image = rgb_image.convert('L')
    gray_image.save('output_gray.tif')
    rgb_image.close()
    return image

def get_pixelvalue(image):
    pixels = list(image.getdata())
    with open('pixel_values.txt', 'w') as file:
        for pixel in pixels:
            # 写入像素值到文本文件
            file.write(f'{pixel}\n')

def produce_img(pixels,width,height):
    new_image = Image.new("L", (width, height))
    new_image.putdata(pixels)
    new_image.save('classify.png')
    return new_image
```

B Kmean 算法函数

```
import numpy as np
from pyspark.sql import Row
from pyspark.sql.functions import col, mean
from pyspark.ml.linalg import Vectors

def kmeans(data_rdd, k = 3, seed = None, epoch = 10):
    # 初始化聚类中心
    np.random.seed(seed)
    length = 1
    lb = 0
    up = 256
    centroids = np.random.randint(lb, up, size=(k, length))


    i = 0
    while i < epoch:
        print('i = %d' % i)
        # 需要将中心点按照大小排列
        centroids =
            np.sort(centroids.flatten()).reshape(centroids.shape)
        # 计算每个样本点到每个聚类中心的距离
        distances = data_rdd.map(lambda point:
            (np.argmin([np.linalg.norm(point - c) for c in
            centroids]), point))

        i += 1
        if i < epoch:
            clusters = distances.groupByKey()
            # 通过groupByKey(对第一列元素group)对distances
                (cluster_index, point)分类，把同一类的点归类
            # 得到的cluster为rdd数据格式，并且有多少类就有多少行，每行第
            #一个元素代表类号，第二个元素是一个迭代器代表该类别中的所有点

```

```
            # 计算新的聚类中心
            new_centroids_rdd = clusters.map(lambda cluster:
                np.round(np.mean(np.array(list(cluster[1])), axis=0)))
            new_centroids = new_centroids_rdd.collect()
            # 通过map读取clusters中每一类中所有点
                np.array(list(cluster[1]))并转换成数组格式,
            # np.mean( ... , axis=0))
                代表计算每一列各自的均值，这里每一列代表每一类
                均值即为新的类中心
            # 最后整数化返回给new_centroids

            # 更新聚类中心
            centroids = np.array(new_centroids)

    # 预测
    predictions = distances.map(lambda x:
        Row(features=Vectors.dense(x[1]), prediction=int(x[0])))
    print(centroids)

    return predictions
```

C 主函数

```
import findspark
findspark.init()

import numpy as np
from pyspark import SparkContext
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import when,col, lit,
    monotonically_increasing_id as mi
from pyspark.ml.clustering import KMeans
from pyspark.ml.linalg import Vectors
import matplotlib.pyplot as plt
```

```
from mpl_toolkits.mplot3d import Axes3D
from Extract_pic import bmp2tif, tif_rgb2gray, get_pixelvalue,
    produce_img
from PIL import Image
from pyspark.ml.evaluation import ClusteringEvaluator
# 引入自己编写的kmean
from Kmean import kmeans

if __name__ == "__main__":

    # process image
    filepath = 'input.bmp'
    bmp2tif(filepath)
    image = tif_rgb2gray()
    get_pixelvalue(image)
    height = image.height
    width = image.width
    image.close()

    # build RDD
    spark = SparkContext(appName="KMeans_pyspark",
        master='local[8]') # SparkContext
    SparkSession(spark)

    # convert data to feature vector
    def f(x):
        rel = {}
        rel['pixel_values'] = Vectors.dense(float(x[0]))
        return rel

    # read data
    data_rdd = spark.textFile('./pixel_values.txt').map(lambda
        line:line.split('\t')).map(lambda p: Row(**f(p)))

    # ----Kmeans聚类----
```

```
    k = 3
    classify_rdd = kmeans(data_rdd, k)


    kmeans_results = classify_rdd.toDF()
    kmeans_results = kmeans_results.withColumn("new_pixel_value",
        lit(0))

    kmeans_results =
        kmeans_results.withColumnRenamed("pixel_values", "features")
    evaluator = ClusteringEvaluator()
    # 设置评估指标为'silhouette'
    silhouette = evaluator.evaluate(kmeans_results)
    print("Silhouette Score:", silhouette)
    kmeans_results = kmeans_results.withColumnRenamed("features",
        "pixel_values")

    new_pixel_value_list = []
    new_pixel_value_list.append(0.0)
    for i in range(1, k-1):
        new_pixel_value_list.append(i*(256.0/(k-1)))
    new_pixel_value_list.append(255.0)

    for cluster in range(0, k):
        new_pv = new_pixel_value_list[cluster]
        kmeans_results = kmeans_results.withColumn("new_pixel_value",
            when(col("prediction") == cluster,
            new_pv).otherwise(col("new_pixel_value")))

    pixels_data = kmeans_results.select("new_pixel_value").collect()
    pixels = [row.new_pixel_value for row in pixels_data]

    produce_img(pixels, width, height)
    new_image = Image.open('classify.png')
    new_image.show()
```

```
    new_image.close()
```

D 调用 MLlib 中的主函数

```
import findspark
findspark.init()

import numpy as np
from pyspark import SparkContext
from pyspark.sql import SparkSession, Row
from pyspark.sql.functions import when, col, lit,
    monotonically_increasing_id as mi
from pyspark.ml.clustering import KMeans
from pyspark.ml.linalg import Vectors
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from Extract_pic import bmp2tif, tif_rgb2gray, get_pixelvalue,
    produce_img
from pyspark.ml.evaluation import ClusteringEvaluator
from PIL import Image

if __name__ == "__main__":

    # process image
    filepath = 'input.bmp'
    bmp2tif(filepath)
    image = tif_rgb2gray()
    get_pixelvalue(image)
    height = image.height
    width = image.width
    image.close()

    # build RDD
    spark = SparkContext(appName="KMeans_pyspark",
```

```
        master='local[2]') # SparkContext
    SparkSession(spark)

    # convert data to feature vector
    def f(x):
        rel = {}
        rel['pixel_values'] = Vectors.dense(float(x[0]))
        return rel

    # read data
    data = spark.textFile('./pixel_values.txt').map(lambda
        line:line.split('\t')).map(lambda p: Row(**f(p))).toDF()
    data.show()

    k = 3
    kmean_model = KMeans(featuresCol='pixel_values').setK(k)
    kmean_run = kmean_model.fit(data)

    kmeans_results = kmean_run.transform(data)
    kmeans_results = kmeans_results.withColumn("new_pixel_value",
        lit(0))

    # 查看聚类中心
    cluster_centers = kmean_run.clusterCenters()
    print("Cluster Centers:")
    for center in cluster_centers:
        print(center)

    kmeans_results =
        kmeans_results.withColumnRenamed("pixel_values", "features")
    evaluator = ClusteringEvaluator()
    # 设置评估指标为'silhouette'
    Silhouette = evaluator.evaluate(kmeans_results)
    print("Silhouette Score:", Silhouette)
    kmeans_results = kmeans_results.withColumnRenamed("features",
```

```
        "pixel_values")

    new_pixel_value_list = []
    new_pixel_value_list.append(0.0)
    for i in range(1, k-1):
        new_pixel_value_list.append(i*(256.0/(k-1)))
    new_pixel_value_list.append(256.0)

    for cluster in range(0, k):
        new_pv = new_pixel_value_list[cluster]
        kmeans_results = kmeans_results.withColumn("new_pixel_value",
            when(col("prediction") == cluster,
            new_pv).otherwise(col("new_pixel_value")))

    pixels_data = kmeans_results.select("new_pixel_value").collect()
    pixels = [row.new_pixel_value for row in pixels_data]

    produce_img(pixels, width, height)
    new_image = Image.open('classify.png')
    new_image.show()
    new_image.close()
```