# 上海财经大学

SHANGHAI UNIVERSITY OF FINANCE AND ECONOMICS

# 《数据科学导论》期末报告

## 题目：基于机器学习的猫狗图片分类模型研究


姓名：谢嘉薪

学号：2020111142

班级：20 数据科学

# 摘要

随着计算机视觉领域的高速发展，深度学习在图片分类、人脸识别、目标检测等任务中表现出色，但传统机器学习方法并非再无意义，基于此背景，结合课上第一次接触到图像数据后对计算机视觉领域的初步认识，本文选取 Kaggle 上的公开动物图片数据集——Animal Faces-HQ(AFHQ)中的 9892 张训练图片数据和 1000 张测试图片数据，其中将猫、狗图片作为响应变量，通过局部二值模式（LBP）提取图片特征，然后分别建立 KNN 模型、logistic 回归模型、SVM 模型进行猫狗图片分类，采取混淆矩阵、ROC 曲线、准确率、召回率、精确率和 F1-score 进行模型分类效果评估。

本文在特征提取上对 LBP 进行了等价模式改进，并对图片灰度值的转换过程可视化，利用基于 SVD 的 PCA 对数据降维，便于后续模型搭建。模型建立阶段，本文对三种模型的原理及求解思路进行了详细阐述，建立 KNN 模型时利用 kd 树提高算法效率，建立 SVM 模型时利用对偶优化和 ADMM 两种算法求解，最后各模型的分类效果优良，本文也提出了不足之处，如改进 LBP 模式、采用 stacking 集成模型、推广到多分类问题上等。

**关键词：**图片分类 局部二值模式 最近邻分类 逻辑回归 支持向量机

# 目录

# 一、引言

## （一）问题综述

计算机视觉是将图像和视频转换成机器可理解的信号的主题。利用这些信号，程序员可以基于这种高级理解来进一步控制机器的行为。在许多计算机视觉任务中，图像分类是最基本的任务之一。它不仅可以用于许多实际产品中，例如 Google Photo 的标签和 AI 内容审核，而且还为许多更高级的视觉任务（例如物体检测和视频理解）打开了一扇门。近几年来，深度学习算法在图像分类方面取得了突破性进展，通过建立层级特征自动提取模型，得到更准确且接近图像高级语义的特征。卷积神经网络（CNN）是一种识别率很高的深度学习模型，能够提取具有平移、缩放、旋转等不变性的结构特征。自 2012 年 ImageNet 大规模视觉识别挑战赛中，Alex Krizhevsky 提出了基于 CNN 的解决方案来应对这一挑战开始，AlexNet 定义了未来十年的实际分类网络框架：卷积，ReLu 非线性激活，MaxPooling 和 Dense 层的组合。

深度学习框架在图像分类任务上的出色表现无需赘述，但传统机器学习方法也并非退出舞台，相反，采用传统图像特征提取方式如 HOG、LBP、SIFT 等，和 logistic、KNN、SVM 等传统机器学习方法，在取得较为良好的结果的同时可以提高模型的可解释性。基于此，本文选取 Kaggle 上公开的猫狗图片数据，利用局部二值模式（LBP）提取图片特征，通过基于 kd 树的 KNN 模型、logistic 模型、采用对偶优化和 ADMM 算法求解的 SVM 模型进行猫狗图片分类，采取混淆矩阵、ROC 曲线、准确率、召回率、精确率和 F1-score 进行模型分类效果评估。

## （二）数据介绍

本文数据集由 Gary Bendig 拍摄，由 Larxel 发布在 Kaggle 平台，数据集名为 Animal Faces-HQ（AFHQ）[1]，该数据集由 16,130 张分辨率为 512*512 的高质量图像组成，已划分出训练集和测试集。本文聚焦于二分类问题，因此使用其中包括 5,153 张猫图片、4,739 张狗图片作为训练集，500 张猫图片、500 张狗图片作为测试集。数据质量高且已打上标签，两类图片数量相近，基本不存在类别不均的情况，便于分类模型的建立。

数据集概览如图 1 所示。

图 1 数据集概览

[1] 数据来源：https://www.kaggle.com/datasets/andrewmvd/animal-faces

# 二、图像处理

众所周知，计算机不认识图像，只认识数字。为了使计算机能够“理解”图像，从而具有真正意义上的“视觉”，本章将尝试从图像中提取有用的数据或信息，得到图像的“非图像”表示或描述，如数值、向量和符号等。这一过程就是特征提取，而提取出来的这些“非图像”的表示或描述就是特征。有了这些数值或向量形式的特征我们就可以通过传统机器学习中的分类模型进行图片分类。本文聚焦 LBP 特征提取模式，并对原始 LBP 模式根据跳变次数降维，即等价模式 LBP 特征，然后对图片矩阵进行降维，保留主要信息的同时减少后续计算量，最后进行数据标准化，利于后续模型建立。

## （一）局部二值模式（LBP）

LBP 指局部二值模式，英文全称：Local Binary Pattern，是一种用来描述图像局部特征的算子，LBP 特征具有灰度不变性和旋转不变性等显著优点。它是由 T. Ojala, M.Pietikäinen, 和 D. Harwood 在 1994 年提出，由于 LBP 特征计算简单、效果较好，因此 LBP 特征在计算机视觉的许多领域都得到了广泛的应用，其中比较出名的应用是人脸识别，可以迁移至类似的猫狗图片分类上。虽然在计算机视觉开源库 Opencv 中已有现成的 LBP 特征提取接口，但为了深入理解图像数据的处理方式，本文从 LBP 原理出发，自行编码实现图片特征的提取。

### 1、原始 LBP 特征计算原理

原始的 LBP 算子定义在每一个非边缘像素点的 3*3 邻域内，以邻域中心像素为阈值，相邻的 8 个像素点的灰度值与邻域中心的像素值进行比较，若周围像素大于中心像素值，则该像素点的位置被标记为 1，否则为 0。这样，3*3 邻域内的 8 个点经过比较可产生 8 位二进制数，将这 8 位二进制数依次排列形成一个二进制数字（通常转换为十进制数即 LBP 码，共 256 种），即得到该窗口中心像素点的 LBP 值，并用这个值来反映该区域的纹理信息，如图 2 所示。

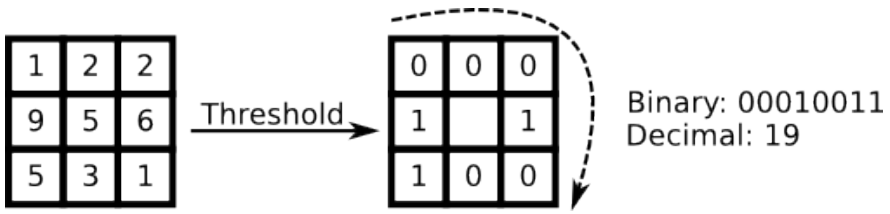


图 2 LBP 图解

基于此，LBP 的操作可被定义为：

$$LBP(x_c, y_c) = \sum_{p=0}^{P-1} 2^p s(I_p - I_c)$$

其中，$(x_c, y_c)$是中心像素，亮度为 $I_c$，而 $I_p(p = 0,1, \dots, P-1)$ 是相邻像素的亮度，$s$定义为:

$$s(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$

2、改进：Uniform Pattern LBP 特征

Uniform Pattern，也被称为等价模式或均匀模式。对输入的每一个图片矩阵，对非边缘像素点作 LBP 处理后，统计该图片矩阵中所有数字出现的频次，返回一个长度为 256 的列表；但随着数据量增大，二进制模式会急剧增多，会使得数据量过大，频次列表过于稀疏，不能很好地反映图像特征。因此，需要对原始的 LBP 模式进行降维，使得数据量减少的情况下能最好的表示图像的信息。

为了解决二进制模式过多的问题，提高统计性，Ojala 提出了采用一种“等价模式”(Uniform Pattern)来对 LBP 算子的模式种类进行降维。在实际图像中，绝大多数 LBP 模式最多只包含两次从 1 到 0 或从 0 到 1 的跳变，且这些序列囊括了“亮点”、“暗点”、“平坦区域”、“变化的边缘”等等，基本包含了图像中绝大部分主要信息。因此只要记录这些二进制序列就行，其余的序列都看成是等价的。

因此，可将“等价模式”定义为：当某个 LBP 所对应的循环二进制数从 0 到 1 或从 1 到 0 最多有两次跳变时，该 LBP 所对应的二进制就称为一个等价模式类。如 00000000（0 次跳变），00000111（只含一次从 0 到 1 的跳变），10001111（先由 1 跳到 0，再由 0 跳到 1，共两次跳变）都是等价模式类。除等价模式类以外的模式都归为另一类，称为混合模式类，例如 10010111（共四次跳变）。通过这样的改进，二进制模式的种类大大减少，而不会丢失任何信息。对于 3×3 邻域内 8 个采样点来说，二进制模式由原始的 256 种减少为 58 种，即：它把值分为 59 类，58 个 uniform pattern 为一类，其它的所有值为第 59 类。这样直方图从原来的 256 维变成 59 维。这使得特征向量的维数更少，并且可以减少高频噪声带来的影响。

具体实现：采样点数目为 8 个，即 LBP 特征值有 28 种，共 256 个值，正好对应灰度图像的 0-255，因此原始的 LBP 特征图像是一幅正常的灰度图像，而等价模式 LBP 特征，根据 0-1 跳变次数，将这 256 个 LBP 特征值分为了 59 类。通过编写字典（uniform_map），将跳变不大于两次的 58 种序列分别映射到 0-57，将其他所有序列模式映射到 58，即等价模式 LBP 特征下，等价模式类在 LBP 特征图像中的灰度值为 0~57，混合模式类在 LBP 特征中的灰度值为 58，灰度值越小颜色越深，因此等价模式 LBP 特征图像整体偏暗。

表 1 uniform_map

| 序列值 | 灰度值 | 序列值 | 灰度值 | 序列值 | 灰度值 | 序列值 | 灰度值 | 序列值 | 灰度值 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 24 | 12 | 112 | 24 | 192 | 36 | 241 | 48 |
| 1 | 1 | 28 | 13 | 120 | 25 | 193 | 37 | 243 | 49 |
| 2 | 2 | 30 | 14 | 124 | 26 | 195 | 38 | 247 | 50 |
| 3 | 3 | 31 | 15 | 126 | 27 | 199 | 39 | 248 | 51 |
| 4 | 4 | 32 | 16 | 127 | 28 | 207 | 40 | 249 | 52 |
| 6 | 5 | 48 | 17 | 128 | 29 | 223 | 41 | 251 | 53 |
| 7 | 6 | 56 | 18 | 129 | 30 | 224 | 42 | 252 | 54 |
| 8 | 7 | 60 | 19 | 131 | 31 | 225 | 43 | 253 | 55 |
| 12 | 8 | 62 | 20 | 135 | 32 | 227 | 44 | 254 | 56 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 14 | 9 | 63 | 21 | 143 | 33 | 231 | 45 | 255 | 57 |
| 15 | 10 | 64 | 22 | 159 | 34 | 239 | 46 | Else | 58 |
| 16 | 11 | 96 | 23 | 191 | 35 | 240 | 47 | | |

### 3、具体实现与示例

编写程序读入 9,892 张图片数据作为训练集和验证集，读入 1,000 张图片作为测试集。由于读取图片时较慢，利用了 tqml 库设置了一个简易进度条能实时刷新进度。

由于数据集中的图片特征集中体现在中心区域，故每张图片在录入时先进行裁剪：将其从中心向外裁剪为 256*256 的大小规模，再将该图像裁剪为 8*8 共 64 个小块（cell）。然后对每一个 cell 采用 LBP 方法提取图像局部特征，最终将每张图片变为 8*8*59 的三维矩阵；所有数据集构成了 9892*8*8*59 的四维矩阵。

取其中一张图片为例，展示图片转换的过程，如图 3 所示。

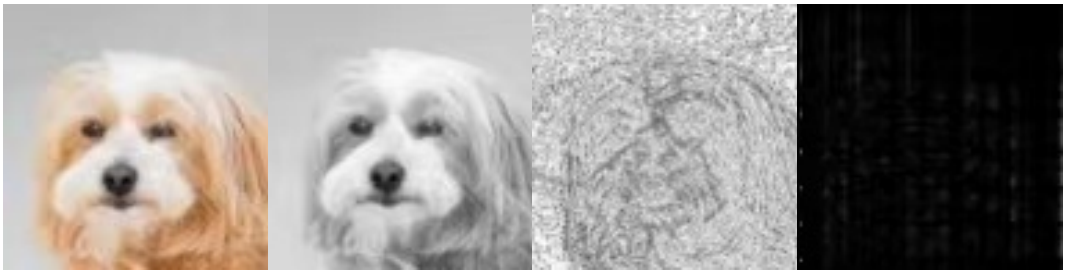

图 3 图片灰度值转换过程

## （二）基于 SVD 的 PCA 降维

所谓降维，即通过某种变化找出原始维度 n 个新的特征向量及其构成的 n 维空间 V，并将原始数据映射到新空间中，进而选取前 k 个信息量最大的特征，实现将 n 维空间降至 k 维。SVD 和 PCA 两种算法均可实现降维，区别仅在于矩阵分解的方法不同、信息量的衡量指标不同。

PCA 使用方差作为信息量的衡量指标，并且通过特征值分解来找出空间 V。降维时，通过产生协方差矩阵 $\frac{1}{n-1}X^TX$，将特征矩阵$X$分解为$Q\Sigma Q^{-1}$，其中$Q$、$Q^{-1}$是辅助的矩阵，Σ是一个对角矩阵，其对角线上的元素就是方差。降维完成之后，PCA 找到的每个新特征向量就叫做“主成分”，而被丢弃的特征向量被认为信息量很少，这些信息很可能就是噪音。

SVD 使用奇异值分解将特征矩阵$X$分解为$U\Sigma V^T$，其中 Σ 是一个对角矩阵，对角线上的元素是奇异值，作为 SVD 中用来衡量特征上的信息量的指标。$U$是经过变换后的新的正交基，称作左奇异矩阵，$V$是原始域的标准正交基，是$X^TX$的特征向量。

由于 PCA 需要先求出协方差矩阵，导致计算量极大，而协方差矩阵的特征向量是 PCA 主成分的方向，SVD 分解得到的$V$又是$X^TX$的特征向量，因此 SVD 的$V$就是协方差矩阵，即 PCA 的主成分方向。至此，两者建立联系，将降维流程拆解为两部分，聚焦于本文的图片数据处理，具体方法如下：

① 利用 SVD 计算特征空间：将输入的图片矩阵（假设大小为 m*n）作 SVD

分解，得到$U(\mathrm{m}*\mathrm{m})$、$\Sigma(\mathrm{m}*\mathrm{n})$、$V(\mathrm{n}*\mathrm{n})$三个矩阵，再根据输入的维数对正交矩阵$V$作切割，取其前 k 行得到大小为(k*n)的矩阵，作为新特征空间。

② 利用 PCA 映射数据和求解新特征矩阵：将新输入的图片矩阵（验证集和测试集）映射到新特征空间上，即$X_{(m,n)}*V_{(k,n)}^{T}$，得到降维后的特征矩阵$X_PCA_{(m,k)}$。

由于得到的奇异值是按大小排序的，越靠前的奇异值包含的信息量越大，因此在将数据降维的情况下依然保留了主要的信息。

### （三）数据标准化

对于通过梯度下降求参数的算法和基于距离的算法来说，标准化或归一化可以加快求最优解的速度并有可能提高精度，由于本文即将建立的模型涉及梯度下降法和距离，利用公式 $X^{*}=\frac{X-\mu}{\sigma}$ 对数据进行标准化，其中$\mu$为数据集的均值，$\sigma$为标准差。

## 三、机器学习模型

### （一）KNN 模型

#### 1、KNN 模型原理

KNN 又称为 k 近邻算法，英文全称为 k-Nearest Neighbor，是机器学习中一种基本的分类与回归算法。KNN 属于一种有监督学习算法，其基本分类思想基于假设：越接近的两个个体，它们的属性就越相似，属于同一类别的可能性也就越高。

据此 k 近邻算法的分类流程如下：对于一个类别未知的样本，基于某种相似性评价策略，从类别已知的训练样本中找出 k 个与待分类样本最相似最接近的样本，对这 k 个样本的类别进行统计，找出样本数最多的那个类别作为待分类样本的最终类别。

k 近邻算法没有一个显示的学习过程，对每一输入的待分类样本点都要依据确定好的 k 值和相似性评价策略（如各种距离度量方法）选出与待分类样本最近的 k 个已标记样本，再依据分类决策规则（如投票策略）确定待分类样本点的类别，每一个输入的待分类样本都有 k 个不同的最近样本点，因此实现 k 近邻算法的难点在于如何快速找到与待分类样本最近的 k 个样本点。

最简单直观的方法就是线性扫描：依据相似性评价策略（以距离为例）计算待分类样本与每一个已标记样本点的距离，依据距离对待分类样本点进行排序，基于排序的结果找出 k 个最近的已标记样本点进行分类。线性扫描的算法复杂度和空间复杂度都较高，当样本数很多、特征空间的维数较多时，线性扫描算法就会非常耗时耗内存，难以满足实际生产的需要。因此需要一种特殊的数据结构存

储训练数据，以减少计算距离的次数，提高算法运行的效率，kd 树就能够满足上述要求。

## 2、kd 树原理及算法实现

kd 树是一种对 n 维空间中的样本点进行划分存储，以便对其进行快速检索的树型数据结构。基于 kd 树算法实现 KNN 模型分为两步：构造 kd 树和搜索 kd 树。

构造 kd 树：利用传入的训练集创建二叉树，深度为$log_2(n+1)$，n 为节点数（即数据量），根据当前所创建树根节点的深度 j 选择第 i=j 个特征为此深度的切分特征（若深度大于特征数，则循环使用下一特征作为切分特征），以训练集中所有实例第 i 个特征的中位数为切分点，即为当前所创建树的根节点。小于该中位数的实例划分至此节点的左子树，传入参数实例构成的新集合与深度 j+1，递归调用该算法继续构造子树；同理大于该中位数的实例划分至此节点的右子树，以新集合和深度 j+1 为参数递归调用该算法构造子树。直到所有新构建节点均为叶节点则结束 kd 树的构造，kd 树中各层节点依次循环使用各特征进行切分，最终把整个特征空间切分成一个个小的超矩形区域。以二维平面为例，构造过程如图 4 所示。

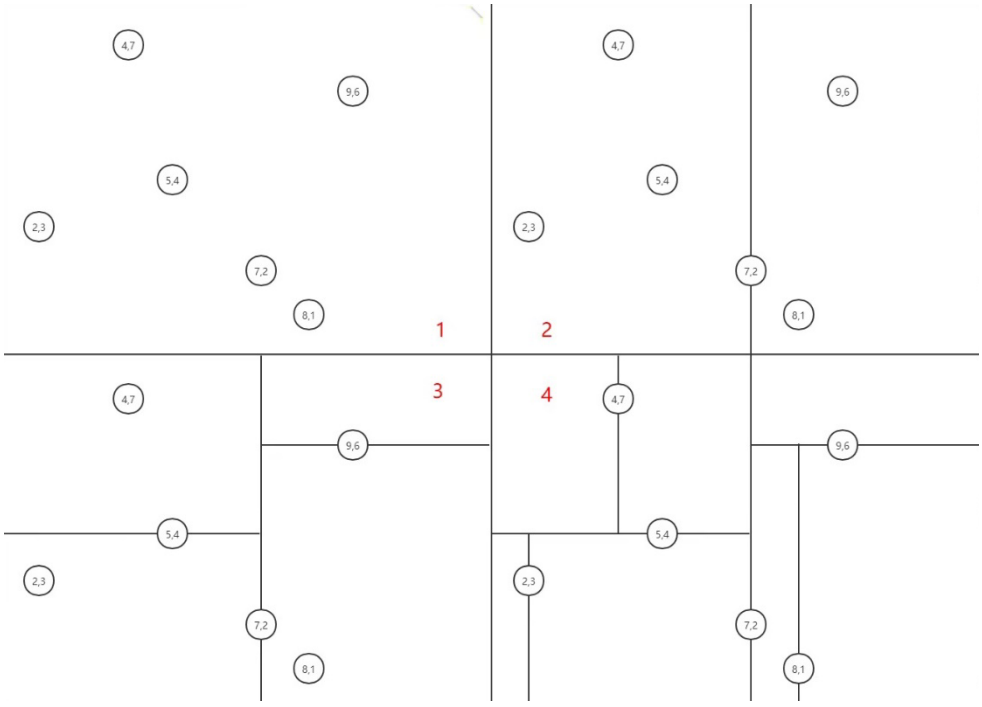


图 4 特征空间划分图解

搜索 kd 树：即在已有 kd 树中搜索与给定的输入实例最近的 k 个邻居。传入参数为输入实例 x、树根节点root、当前最近邻居$X_{min}$和当前最近距离$R_{min}$。易知每个叶节点对应最小切分区域，因此给定的输入实例必然位于某个叶节点对应的区域内，所以首先找到输入实例对应的叶节点。从根节点出发递归向下访问 kd 树，即判断实例与节点第 i 个特征的大小，若实例小则移动至该节点的左儿子节点，反之移动至右儿子节点，直至找到该叶节点。计算输入实例与叶节点的距离 r，如果$r < R_{min}$，则更新$X_{min}$和$R_{min}$。递归向根节点回退，每次搜索父节点对应的区域，判断是否要移动至兄弟节点：若实例 x 与父节点的距离$r < R_{min}$则更新

$X_{min}$和$R_{min}$，判断以 x 为球心，$R_{min}$为半径的超球体是否与兄弟节点对应的区域相交，即比较实例与兄弟节点的距离和$R_{min}$的大小，若相交则以兄弟节点为根递归调用算法尝试更新$X_{min}$和$R_{min}$。当回退到根节点时算法结束，返回最近邻居$X_{min}$和最近距离$R_{min}$。

以上为最邻近搜索，使用一个容量为 k 的最大堆存储 k 个最近邻居以及相应的 k 个距离即可将该算法推广至 k 邻近搜索。

### 3、KNN 模型建立及效果评估

将数据集按 7:3 划分为训练集和验证集，取最近邻居数 k 为 1~20，利用训练集构造的 kd 树对验证集中的每一个实例进行预测，取准确率最高时的 k 值为最优参数，进而利用全体训练集和验证集构造最优 kd 树，对测试集进行预测。

通过训练集和验证集得到最优参数为：k=6，根据图 5 可以发现当 k 超过一定数量后模型效果显著下降。KNN 模型的预测混淆矩阵及各项评价指标如表 2 所示，ROC 曲线如图 6 所示，此时 AUC 为 0.958，分类效果良好。

表 2 KNN 模型混淆矩阵及各项评价指标

| 混淆矩阵 | | 真实值 | |
|---|---|---|---|
| | | 1 | 0 |
| 预测值 | 1 | 479 | 21 |
| | 0 | 21 | 479 |

| | |
|---|---|
| 准确率 | 0.958 |
| 召回率 | 0.958 |
| 精确率 | 0.958 |
| F1-score | 0.958 |

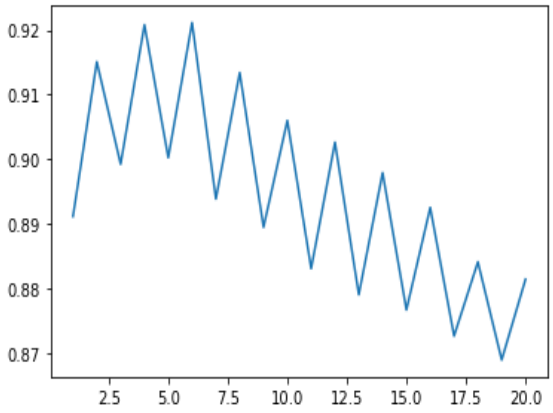


图 5 验证集预测准确率随 k 值的变化

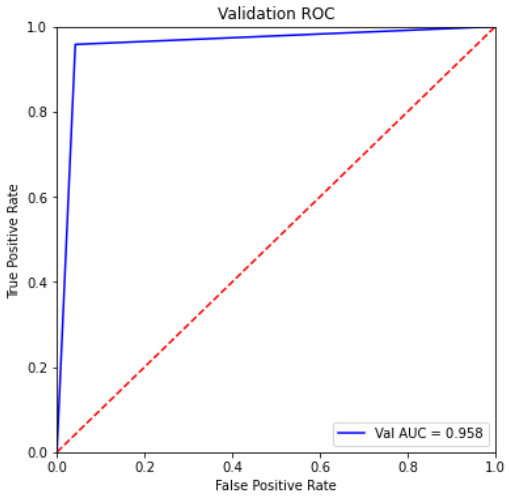


图 6 KNN 模型 ROC 曲线

## （二）Logistic 回归模型

### 1、Logistic 回归模型原理及梯度下降实现

假设有数据集$\{(y_i, x_i)\}_{i=1}^{n}$，其中$y_i \in \{0,1\}$，Logistic 回归尝试估计$p(y_i = 1|x_i) \triangleq p(x_i)$。考虑线性组合$w^T x_i + b$，其中$w$为与图片特征向量同维数的向量，$b$为偏置；为了方便，将输入的图片向量扩增一列，且所有图片样本该列之值均

置为 1；实际上等价于将偏置$b$当作$w_0$来处理。

一个实际的问题是：$p(x_i)$的取值范围为(0,1)，而$w^T x_i + b$可能取到一切实数，即它们的值域不同，考虑 Logit 变换：

$$logit\big(p(x_i)\big) = \log\frac{p(x_i)}{1-p(x_i)}, i = 1, \dots, n$$

即可将函数$\sigma(z) = \frac{1}{1+e^{-z}}$ 作为分类函数，将线性判别式的结果输入分类函数，输出大于 0.5 的实例预测为 1，反之预测为 0。

对于参数向量 w，本文采用梯度下降法求解，基本思路如下：

Step1：将参数向量 w 随机取值，将损失值置为无穷；

Step2：使用当前参数向量 w 进行分类预测，将预测结果和正确结果输入损失函数计算损失；

Step3：对损失函数关于参数 w 求梯度，之后沿着梯度的反方向，根据学习率对参数进行更新；

Step4:，若损失函数减小量小于给定的阈值，停止训练，反之重复 Step2-4。

其中计算损失函数时，采用极大似然法：所有样本（n 个）全部预测正确的概率可用似然函数表示：

$$L(w) = \prod_{i=1}^{n} p(x_i)^{y_i}\big(1-p(x_i)\big)^{1-y_i}$$

则可定义损失函数$Loss(w) = -\frac{L(w)}{n}$，求损失函数的最小值即求似然函数的最大值；求最大值时涉及求梯度操作，对于连乘函数求梯度很复杂，定义对数似然函数：

$$l(w) = \ln(L(w))$$

由于 ln 是单增函数，因此不影响我们的操作；而这一变换之后连乘操作转化为了连加，求梯度便容易得多。最终可求得梯度表达式为：

$$\nabla Loss(w) = \frac{\sum_{i=1}^{n}(p(x_i) - y_i)x_i}{n}$$

## 2、Logistic 回归模型建立及效果评估

将数据集按 7:3 划分为训练集和验证集，取学习率为（0.1, 0.5, 1），取损失减少量阈值为（0.001, 0.0001），利用训练集训练模型并对验证集中的每一个实例进行预测，取准确率最高时的学习率和阈值为最优参数，进而利用全体训练集和验证集训练模型，对测试集进行预测。

通过训练集和验证集得到最优学习率为 1、最优阈值为 0.0001，此时 Logistic 模型的预测混淆矩阵及各项评价指标如表 3 所示，ROC 曲线如图 7 所示，此时 AUC 为 0.996，分类效果极佳。

表 3 Logistic 模型混淆矩阵及各项评价指标

| 混淆矩阵 | | 真实值 | |
|---|---|---|---|
| | | 1 | 0 |
| 预测值 | 1 | 491 | 22 |
| | 0 | 9 | 478 |

| | |
|---|---|
| 准确率 | 0.969 |
| 召回率 | 0.982 |
| 精确率 | 0.957 |
| F1-score | 0.969 |

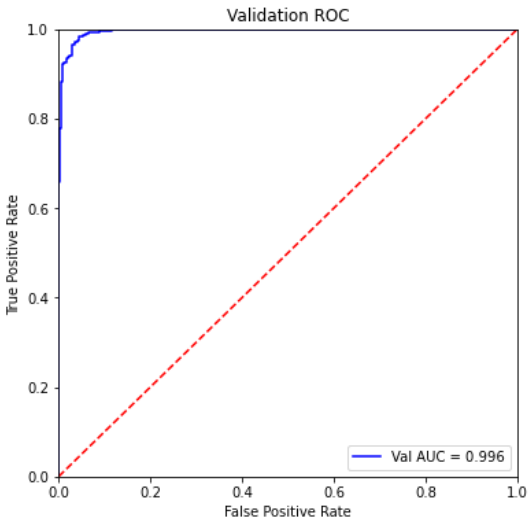


图 7 Logistic 模型 ROC 曲线

## （三）SVM 模型

### 1、SVM 模型原理

对于数据集$\{(x_i, y_i)\}_{i=1}^{n}$，其中$y_i \in \{-1,1\}$，定义分割超平面$\{x: f(x) = \beta^T x + \beta_0 = 0\}$，对于线性可分的数据集来说，存在无数个超平面可以完全划分数据样本，但只有唯一一个分割超平面能使几何间隔最大。定义分类决策边界为$G(x) = sign\big(f(x)\big) = sign(\beta^T x + \beta_0)$，如果样本点$(x_i, y_i)$被错分，则有$y_i f(x_i) < 0$。

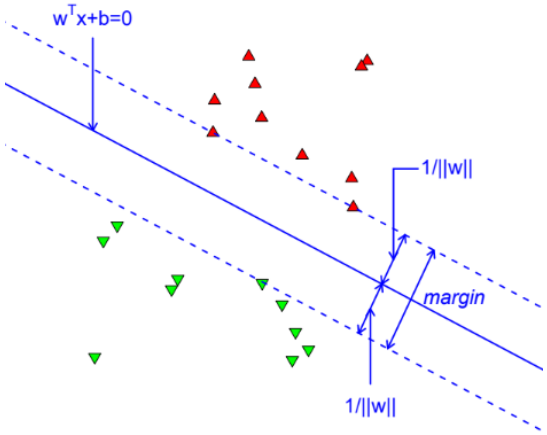


图 8 支持向量机示意图

如图 8 所示，假设图中两条超平面分别代表$\beta^T x + \beta_0 = \pm 1$，两条超平面间的距离为 2C，样本$x_i$到平面$\beta^T x + \beta_0$的距离为$\frac{\beta^T x + \beta_0}{||\beta||}$，为了使距离最大，即求：

$$\max C$$

$$s.t.\frac{1}{||\beta||}y_i(\beta^T x_i+\beta_0)\geq 1; i=1,\dots,n$$

令$C=\frac{1}{||\beta||}$，则上述问题等价于：

$$min\frac{1}{2}||\beta||^2$$

$$s.t.\, y_i(\beta^T x_i+\beta_0)\geq 1; i=1,\dots,n$$

因此，上式拉格朗日函数可以写为：

$$\mathcal{L}_P=min\frac{1}{2}||\beta||^2-\sum_{i=1}^{n}\alpha_i(y_i(x_i^T\beta+\beta_0)-1), s.t.\,\alpha_i\geq 0$$

对$\beta,\beta_0$求导并令其等于 0，得到：

$$\begin{cases}\beta=\sum_{i=1}^{n}\alpha_i y_i x_i\\ 0=\sum_{i=1}^{n}\alpha_i y_i\end{cases}$$

将结果代入到$\mathcal{L}_P$，得到 Wolfe 对偶形式：

$$max\mathcal{L}_D(\alpha)=\max(\sum_{i=1}^{n}\alpha_i-\frac{1}{2}\sum_{i=1}^{n}\sum_{i'=1}^{n}\alpha_i\alpha_{i'}y_i y_{i'}<x_i,x_{i'}>)$$

$$s.t.\,\alpha_i\geq 0\ \ and\ \ \sum_{i=1}^{n}\alpha_i y_i=0$$

而在处理实际问题时，数据常常不可分，可以引入松弛变量$\xi_i$来放宽约束条件，则优化问题可进一步转化为：

$$min\frac{1}{2}||\beta||^2+C\sum_{i=1}^{n}\xi_i$$

$$s.t.\begin{cases}y_i(\beta^T x_i+\beta_0)\geq 1-\xi_i\\ \xi_i\geq 0\end{cases}; i=1,\dots$$

考虑上式中有关$\xi_i$的项，上式可改写为：

$$min\sum_{i=1}^{n}\max(0,1-y_i(\beta^T x_i+\beta_0))+\frac{\lambda}{2}||\beta||^2$$

则拉格朗日函数为：

$$\mathcal{L}_P(\beta,\beta_0,\xi)=\frac{1}{2}||\beta||^2+C\sum_{i=1}^{n}\xi_i-\sum_{i=1}^{n}\alpha_i(y_i(\beta^T x_i+\beta_0)-1+\xi_i)-\sum_{i=1}^{n}\gamma_i\xi_i$$

其中$\alpha_i,\xi_i,\gamma_i\geq 0$

对$\mathcal{L}_P$求一阶导，并令其等于 0：

$$\begin{cases} \dfrac{\partial \mathcal{L}_P}{\partial \beta} = \beta - \sum_{i=1}^{n} \alpha_i y_i x_i = 0 \\ \dfrac{\partial \mathcal{L}_P}{\partial \beta_0} = \sum_{i=1}^{n} \alpha_i y_i = 0 \\ \dfrac{\partial \mathcal{L}_P}{\partial \xi_i} = C - \alpha_i - \xi_i = 0 \end{cases}$$

解得：

$$\begin{cases} \beta = \sum_{i=1}^{n} \alpha_i y_i x_i \\ \sum_{i=1}^{n} \alpha_i y_i = 0 \\ C = \alpha_i + \xi_i \end{cases}$$

将上述结果代回到拉格朗日函数中，可得：

$$\mathcal{L}_P = \frac{1}{2}\sum_{i=1}^{n}\sum_{i\prime=1}^{n} \alpha_i \alpha_{i'} y_i y_{i'} x_i^T x_{i'} + \sum_{i=1}^{n} \alpha_i + C\sum_{i=1}^{n} \xi_i - \sum_{i=1}^{n} \xi_i(\alpha_i + \gamma_i) - \sum_{i=1}^{n} \alpha_i y_i x_i^T \beta$$

$$= \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{i'=1}^{n} \alpha_i \alpha_{i'} y_i y_{i'} x_i^T x_{i'}$$

于是该优化问题可转化为对偶问题：

$$max\mathcal{L}_D(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{i'=1}^{n} \alpha_i \alpha_{i'} y_i y_{i'} x_i^T x_{i'}$$

$$s.t. \sum_{i=1}^{n} \alpha_i y_i = 0, \qquad 0 \le \alpha_i \le C$$

## 2、使用对偶优化求解支持向量机

进一步将对偶优化问题写成向量化的形式：

$$maxL_D(\alpha) = \alpha - \frac{1}{2}\alpha^T (yy^T)(xx^T)\alpha$$

利用 cvxopt.solvers 函数求解二次型，二次规划问题标准形式为：

$$min\frac{1}{2}X^T PX + q^T X$$

$$s.t. Gx \le h, AX = b$$

因此，可将上述问题改写成：

$$min\frac{1}{2}\alpha^T P\alpha + q^T \alpha$$

$$s.t. Gx \le h, AX = b$$

其中：

$$\begin{cases} P = (yy^T)(xx^T) \\ q = -1 \\ G = \begin{pmatrix} I_n \\ -I_n \end{pmatrix} \\ h = (c, \dots, c, 0, \dots, 0)^T \\ A = y^T \\ b = 0 \end{cases}$$

由此可解出$\alpha$，通过$\beta = \sum_{i=1}^{n} \alpha_i y_i x_i = X^T \alpha y$（向量形式），可以求得超平面的系数 beta，通过$y_i\left(\hat{\beta}^T x_i + \beta_0\right) = 1$，可以求得超平面的截距项为：

$$\beta_0 = \frac{1}{y} - X\hat{\beta}^T$$

### 3、使用 ADMM 算法求解支持向量机

考虑 SVM 的标准正则化形式：

$$min \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i \beta^T x_i) + \frac{\lambda}{2}\|\beta\|_2^2$$

引入$\alpha_i = \max(0, 1 - y_i \beta^T x_i)$，上述问题变为：

$$min \frac{1}{n}\sum_{i=1}^{n} \alpha_i + \frac{\lambda}{2}\|\beta\|_2^2$$

$$s.t. 1 - y_i \beta^T x_i = \alpha_i - \xi_i, \alpha_i \geq 0, \xi_i \geq 0$$

上述问题的增广拉格朗日函数为：

$$L(\beta, \alpha, \xi, \mu) = \frac{1}{n}\sum_{i=1}^{n} \alpha_i + \frac{\lambda}{2}\|\beta\|_2^2 + \frac{\rho}{2}\sum_{i=1}^{n}(1 - y_i \beta^T x_i - \alpha_i + \xi_i + \mu_i)^2$$

具体迭代步骤为：

$$\beta^{t+1} = argmin \frac{\lambda}{2}\|\beta\|_2^2 + \frac{\rho}{2}\sum_{i=1}^{n}(1 - y_i \beta^T x_i - \alpha_i^t + \xi_i^t + \mu_i^t)^2$$

$$(\alpha_i^{t+1}, \xi_i^{t+1}) = argmin \frac{1}{n}\sum_{i=1}^{n} \alpha_i + \frac{\rho}{2}\sum_{i=1}^{n}(1 - y_i x_i^T \beta^{t+1} - \alpha_i + \xi_i + \mu_i^t)^2$$

$$\alpha_i \geq 0, \xi_i \geq 0$$

$$\mu_i^{t+1} = \mu_i^t + 1 - y_i \beta^T x_i$$

对$\beta^{t+1}$求一阶导数，并根据$\alpha_i \geq 0, \xi_i \geq 0$的约束，对$\alpha_i, \xi_i$求解再做截断，得到如下 ADMM 优化迭代步骤：

$$\beta^{t+1} = (\lambda I_n + \rho X^T X)^{-1} \rho X^T (y(1 - \alpha_i^t + \xi_i^t + \mu_i^t))$$

$$\alpha_i^{t+1} = \max\{1 - y_i x_i^T \beta^{t+1} + \mu_i^t - \frac{1}{n\rho}, 0\}$$

$$\xi_i^{t+1} = \max\{-(1 - y_i x_i^T \beta^{t+1}) - \mu_i^t, 0\}$$

$$\mu_i^{t+1} = \mu_i^t + 1 - y_i \beta^T x_i$$

## 4、SVM 模型建立及效果评估

对偶优化是将 SVM 问题转化为标准 QP 问题，利用 cvxopt.solvers.qp 函数求解二次型，其中参数 C 在$[0.1, 0.5, 1, 5, 10]$中选取，通过将样本按照 9:1 的比例，随机抽样分为训练集和验证集，进行 10 折交叉验证，经准确度比较得出 C=0.1 为较优值。取全体训练集和验证集，将 C 代入 qp 函数求解$\alpha$，再求解超平面的系数和截距项，训练 SVM 并对测试集进行预测，得到的混淆矩阵及各项评价指标如表 4 所示，ROC 曲线如图 9 所示，此时 AUC 为 0.996，模型拟合效果极佳。

表 4 对偶优化求解的 SVM 模型混淆矩阵及各项评价指标

| 混淆矩阵 | | 真实值 | |
|---|---|---|---|
| | | 1 | 0 |
| 预测值 | 1 | 492 | 23 |
| | 0 | 8 | 477 |

| | |
|---|---|
| 准确率 | 0.969 |
| 召回率 | 0.984 |
| 精确率 | 0.955 |
| F1-score | 0.969 |

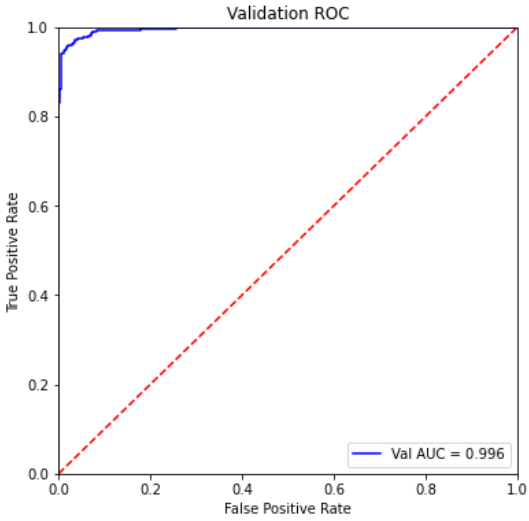


图 9 对偶优化求解 SVM 模型 ROC 曲线

ADMM 算法是机器学习中使用较为广泛的最优化约束方法，将全局问题分解为多个局部子问题，进而求得全局解。ADMM 算法中，对于参数$\lambda$的确定，采用网格筛选的思路，即从$10^{-3+0.1s}, s = 0, \ldots, 60$中选取，将样本按照 9:1 的比例，随机抽样分为训练集和验证集，进行 10 折交叉验证，经准确度比较得出$s = 29$为较优值。取全体训练集和验证集，将$s = 29$代入训练 SVM 模型并对测试集进行预测，得到的混淆矩阵及各项评价指标如表 5 所示，ROC 曲线如图 10 所示，此时 AUC 为 0.993，模型拟合效果较好。

表 5 ADMM 求解的 SVM 模型混淆矩阵及各项评价指标

| 混淆矩阵 | | 真实值 | |
|---|---|---|---|
| | | 1 | 0 |
| 预测值 | 1 | 479 | 27 |
| | 0 | 21 | 473 |

| | |
|---|---|
| 准确率 | 0.952 |
| 召回率 | 0.958 |
| 精确率 | 0.947 |
| F1-score | 0.952 |

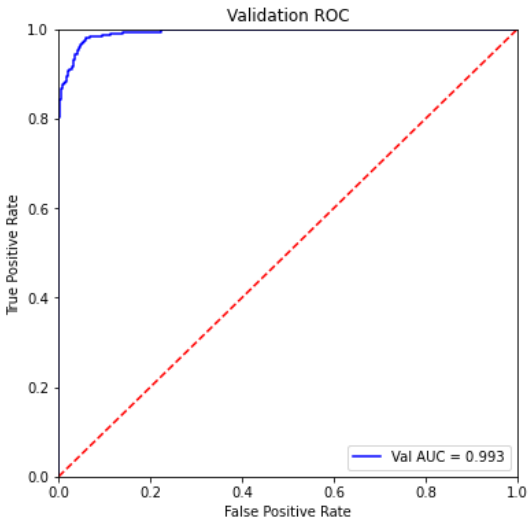


图 10 ADMM 求解 SVM 模型 ROC 曲线

# 四、结论

## （一）模型比较

三种模型的评价指标表如表 6 所示：

表 6 模型评价指标表

| 模型 | 算法 | AUC | 准确率 | 召回率 | 精确率 | F1-score |
|---|---|---|---|---|---|---|
| KNN | kd 树 | 0.958 | 0.958 | 0.958 | 0.958 | 0.958 |
| Logistic | 梯度下降 | **0.996** | **0.969** | 0.982 | **0.957** | **0.969** |
| SVM | 对偶优化 | **0.996** | **0.969** | **0.984** | 0.955 | **0.969** |
| SVM | ADMM | 0.993 | 0.952 | 0.958 | 0.947 | 0.952 |

有如下结论：

① 对于分类问题，AUC 可以反映分类器的优良性，可以发现以上几种模型的分类效果均较高，其中 Logistic 和 SVM 更是逼近 1，这一方面是模型自身的优势，另一方面与所选数据集也有关系，在本文中，主要目的是对猫狗进行分类，这属于计算机视觉领域最基础的图片分类任务，因此取得如此优异成绩不足为奇。

② 结合程序运行时间来看，Logistic 耗时最短，SVM 的两种求解算法中均使用了 10 折交叉验证，虽然选取出了较优参数，但时间成本巨大，这与数据量和数据维度有一定关系。

总体而言，由于数据集质量较高，且不存在类别不均的情况，各分类模型均取得了较好的成绩，其中 Logistic 表现尤为突出。可以看出，对于图片分类任务，利用传统特征提取方式和机器学习方法也能达到一定的准确性。

## （二）待改进之处

① 对于 LBP 特征提取模式，除了改进为等价模式外，还可以改进为圆形 LBP 特征、旋转不变 LBP 特征等，相比于 Opencv 自带的 LBP 处理接口，自行编写的代码存在可视化效果差、处理速度较慢的缺点。对于图片特征提取方式，除了 LBP 外，还可以尝试 HOG、SIFT 等提取方式。

② 虽然本文研究了 KNN、Logistic、SVM 三种模型的分类效果，但采用的都是单一模型采用的是单一模型，可以尝试借助 Stacking 集成方法融合几种模型，来构建新的组合模型，探究集成模型的效果是否有显著提升。

③ 本文仅研究了二分类问题，但可以通过 OvO、OvR、MvM 三种策略将二分类模型推广到多分类问题上。

④ 可以运用卷积神经网络等深度学习方法进行图片分类，并与本文所用方法进行比较。

# 参考文献

[1]姚立平,潘中良.基于改进的 HOG 和 LBP 算法的人脸识别方法研究[J].光电子技术,2020,40(02):114-118+124.DOI:10.19453/j.cnki.1005-488x.2020.02.008.

[2]赵腾浩,杨立娟,王宇,李晶.基于分块 LBP 特征提取和改进 KNN 的手写数字识别[J].信息与电脑(理论版),2021,33(05):195-200.

[3]贾锋,王高,师钰璋,付雨泽.基于改进的 LBP 及 KNN 算法的表情识别[J].国外电子测量技术,2020,39(08):40-44.DOI:10.19652/j.cnki.femt.2002120.

[4]刘吉安,江金滚.基于区域改进 LBP 和 KNN 的人脸识别[J].电脑知识与技术,2016,12(13):184-185.DOI:10.14004/j.cnki.ckt.2016.1857.

[5]祁亨年.支持向量机及其应用研究综述[J].计算机工程,2004(10):6-9.

# 代码附录

```
# 导入相关库
import numpy as np  # 用于进行矩阵计算
import pandas as pd
import matplotlib.pyplot as plt
import cv2  # 用于图片处理
import os
import random
from collections import Counter
from tqdm import tqdm#输出进度条
from queue import deque
from colorama import Fore
from sklearn.metrics import roc_curve,auc
from cvxopt import matrix,solvers #二次型求解
# 图像处理
## LBP
#各种二进制序列的对应表
uniform_map = {0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 6: 5, 7: 6, 8: 7, 12: 8,14: 9, 15: 10, 16: 11, 24: 12, 28: 13, 30: 14, 31: 15, 32: 16, 48: 17,
 56: 18, 60: 19, 62: 20, 63: 21, 64: 22, 96: 23, 112: 24,120: 25, 124: 26, 126: 27, 127: 28, 128: 29, 129: 30, 131: 31, 135: 32,143: 33,
 159: 34, 191: 35, 192: 36, 193: 37, 195: 38, 199: 39, 207: 40,223: 41, 224: 42, 225: 43, 227: 44, 231: 45, 239: 46, 240: 47, 241: 48,
243: 49, 247: 50, 248: 51, 249: 52, 251: 53, 252: 54, 253: 55, 254: 56,255: 57}
'''LBP_uniform 模型，用于提取图片特征'''
class LBP_uniform:
    def __init__(self):
        #最终存储数据的列表
        self.cell_lbp=[]

    #获取一个列表中各个数字出现的频率
    def get_count(self,LIST):
        #获取一个列表中各个数字出现的频率
        count = Counter(LIST)
        #获取一个长度为 59 的整数序列
        a=np.arange(59)
        #将其转化为字典的形式
        count_dict = dict(count)
        def check(x):
            if(x in count_dict.keys()):
                return(count_dict[x])
            else:
                return(0)
        #转化为列表输出
        return (list(map(check,a)))
    #在样本点周围 8 个方向取样，返回一个二进制序列
    def cal_basic_lbp(self,img,i,j):
        sum = []
        if img[i - 1, j ] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i - 1, j+1 ] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i , j + 1] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i + 1, j+1 ] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i + 1, j ] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i + 1, j - 1] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i , j - 1] > img[i, j]:
            sum.append(1)
        else:
            sum.append(0)
        if img[i - 1, j - 1] > img[i, j]:
            sum.append(1)
```

```
            else:
                sum.append(0)
            return sum
        #输入一个 cell，返回该 cell 中所有图样出现的频次
        def fit(self,img):
            self.cell_lbp=[]
            for i in range(1,img.shape[0]-1):
                for j in range(1,img.shape[1]-1):
                    #对非边缘的每一个像素点获得二进制序列
                    binary = self.cal_basic_lbp(img,i,j)
                    #获得二进制数中跳变次数
                    change=0
                    for k in range(len(binary)-1):
                        if(binary[k]!=binary[k+1]):
                            change+=1
                    #如果跳变次数小于 2 则转化为十进制，根据对应表返回相应的序号值
                    if change <= 2:
                        dec = 0#储存计算出的十进制数
                        bit_num = 0 #左移位数
                        for x in binary[::-1]:#从低位读向高位
                            dec += x << bit_num  # 左移 n 位相当于乘以 2 的 n 次方
                            bit_num += 1
                        self.cell_lbp.append(uniform_map[dec])
                    #如果跳变次数大于 2 则返回 58
                    else:
                        self.cell_lbp.append(58)
            cell_list=self.get_count(self.cell_lbp)
            return cell_list
## 降维
'''主成分分析，用于给数据降维'''
class PCA:
    def __init__(self,n_component):
        self.n_component=n_component# 压缩后的维数
        self.V_cut=None#分解出的正交矩阵

    def fit(self,X):
        #普通 SVD 分解
        U,sigma,V=np.linalg.svd(X)
        #根据输入的 n 截取正交矩阵 V
        self.V_cut=V[0:self.n_component,:]
        return np.dot(X,self.V_cut.T)

    #利用从训练集得出的转化矩阵转化测试集和验证集，保证转化方式相同
    def transform(self,X):
        return np.dot(X,self.V_cut.T)
## 功能函数
'''用于对数据进行标准化处理'''
def Std(X):
    #计算均值
    X_mean = X.mean(axis=0)
    # 计算标准差
    X_std = X.std(axis=0)
    # 标准化
    X1 = (X-X_mean)/X_std
    return (X1)

'''用于对图片预处理后再输入到模型中'''
def img_preprocess(img,lbp):
    img_pro=[]#储存该图片每个 cell 中各个样式出现的频次
    img = img[128:384, 128:384]  # 裁剪图片为 98*98，便于裁剪为小块
    a=8#每行/列 cell 个数
    width=int(img.shape[0]/a)
    for row in range(a):
        for col in range(a):
            img_cell=img[row*width:(row+1)*width,col*width:(col+1)*width]#裁剪 cell
```

```
img_pro.append(lbp.fit(img_cell))
    img_pro=np.array(img_pro)
    return img_pro

'''用于根据预测出的结果和标签计算正确率和混淆矩阵'''
def Accuracy(y_true,y_pred):
    TP,FN,FP,TN = 0,0,0,0#分别记录模型预测的结果中的真正，假负，假正，真负类
    for i in range(len(y_pred)):
        if (y_true[i] == 1):
            if (y_pred[i] == 1):
                TP += 1
            elif (y_pred[i] == 0):
                FN +=1
        elif (y_true[i] == 0):
            if (y_pred[i] == 1):
                FP += 1
            elif (y_pred[i] == 0):
                TN +=1

confusion_matrix=np.mat([[TP,FP],[FN,TN]])#计算预测结果的混淆矩阵
    Accuracy = (TP+TN)/(TP+TN+FN+FP)
    Recall=TP/(FN+TP)
    Precision=TP/(TP+FP)
    Specificity=TN/(TN+FP)
    F1_score = (2*Precision*Recall)/(Precision+Recall)
    print("Confusion Matrix:\n",confusion_matrix)
    print("Accuracy=",Accuracy)
    print("Recall=",Recall)
    print("Precision=",Precision)
    print("Specificity=",Specificity)
    print("F1_score=",F1_score)
    return Accuracy
'''ROC'''
def ROCplot(y_true,y_pred):
    fpr,tpr,threshold = roc_curve(y_true, y_pred)
    roc_auc =auc(fpr, tpr)
    plt.figure(figsize=(6,6))
    plt.title('Validation ROC')
    plt.plot(fpr, tpr, 'b', label = 'Val AUC = %0.3f' % roc_auc)
    plt.legend(loc = 'lower right')
    plt.plot([0, 1], [0, 1],'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')
    plt.show()
    print("auc :",roc_auc)
# 读取训练数据
'''加载训练数据'''
print('Loading data...')
num_total = 0  # 总图片数
image = []  # 用于记录图片矩阵的列表
label = []  # 用于记录标签的列表

lbp=LBP_uniform()

content = os.listdir('afhq/train/cat/') # 取 cat 图片
pbar=tqdm(range(len(content)),bar_format='{l_bar}%s{bar}%s{r_bar}' % (Fore.BLUE, Fore.RESET))#设置进度条
for t in pbar:
    label.append(1)  # 标签记为 1
    img = cv2.imread('afhq/train/cat/' + content[t], cv2.IMREAD_GRAYSCALE)
    image.append(img_preprocess(img,lbp))
# 读入处理后的灰度图片
    num_total += 1
    pbar.set_description("cat data")
pbar.close()

content = os.listdir('afhq/train/dog/')  # 取 dog 图片
pbar=tqdm(range(len(content)),bar_format='{l_bar}%s{bar}%s{r_bar}' % (Fore.BLUE, Fore.RESET))#设置进度条
for t in pbar:
    label.append(0)  # 标签记为 0
    img = cv2.imread('afhq/train/dog/' + content[t], cv2.IMREAD_GRAYSCALE)
    image.append(img_preprocess(img,lbp))
```

```
# 读入处理后的灰度图片
        num_total += 1
        pbar.set_description("dog data")
pbar.close()

'''加载测试数据'''
image_test = []  # 用于记录图片矩阵的列表
label_test = []  # 用于记录标签的列表
content = os.listdir('afhq/val/cat/')# 取 cat 图片
pbar=tqdm(range(len(content)),bar_format='{l_bar}%s{bar}%s{r_bar}' % (Fore.BLUE, Fore.RESET))#设置进度条
for t in pbar:
        label_test.append(1)  # 标签记为 1
        img = cv2.imread('afhq/val/cat/' + content[t], cv2.IMREAD_GRAYSCALE)

image_test.append(img_preprocess(img,lbp))
# 读入处理后的灰度图片
        pbar.set_description("cat data")
pbar.close()
content = os.listdir('afhq/val/dog/')  # 取 dog 图片
pbar=tqdm(range(len(content)),bar_format='{l_bar}%s{bar}%s{r_bar}' % (Fore.BLUE, Fore.RESET))#设置进度条
for t in pbar:
        label_test.append(0)  # 标签记为 0
        img = cv2.imread('afhq/val/dog/' + content[t], cv2.IMREAD_GRAYSCALE)

image_test.append(img_preprocess(img,lbp))
# 读入处理后的灰度图片
        pbar.set_description("dog data")
pbar.close()
# Logistic
'''逻辑回归模型，用于最后的输出'''
class LogisticRegression:
    def __init__(self, lr=1e-3, tol=None):
        self.iter_max=2000#最大迭代次数
        self.lr = lr#学习率
        self.tol = tol#误差阈值
        self.w = None#初始化权值向量

    #预测单个实例
    def pred(self, X, w):
        z = np.dot(X,w)#获得线性函数的输出值
        return (1. / (1. + np.exp(-z)))# 用 sigmoid 函数作为输出

    def X_expand(self, X):
        #在 X 后添加一列并置 1，便于和偏置 b 相乘
        X_expand = np.empty((X.shape[0], X.shape[1] + 1))
        X_expand[:, 0:X.shape[1]] = X
        X_expand[:, X.shape[1]] = 1
        return X_expand

    def train(self, X_train, y_train):
        '''训练'''
        # 预处理 X_train(添加 x0=1，用于与偏置 b 相乘)
        X_train = self.X_expand(X_train)
        # 随机初始化 w，便于快速收敛
        self.w = np.random.random(X_train.shape[1])
        # 初始化损失值为正无穷
        loss_old = np.inf

        # 使用梯度下降更新 w
        for step in range(self.iter_max):
            # 预测
            y_pred = self.pred(X_train, self.w)
            # 计算损失
            p = y_pred * (2 * y_train - 1) + (1 - y_train)
            loss= -np.sum(np.log(p)) / y_train.size
            # 如果损失下降不足阈值，则终止迭代
            if loss_old - loss < self.tol:
                break
            #更新损失
            loss_old = loss
            # 计算梯度
```

```
            grad=np.matmul(y_pred - y_train, X_train) / y_train.size
            # 向梯度的反方向更新参数 w
            self.w -= self.lr * grad

    def predict(self, X):
        '''预测'''
        # 预处理 X_test(添加 x0=1)
        X = self.X_expand(X)
        # 预测
        y_pred = self.pred(X, self.w)

        return y_pred
#随机打乱训练集
np.random.seed(2020111142)
y = np.array(label)  # y 表示标签集
X = np.array(image)  # X 表示数据集
X = X.reshape(X.shape[0], X.shape[1] * X.shape[2])#展平 X 中的矩阵
Index = np.arange(num_total)#获取序列
np.random.shuffle(Index)#随机打乱索引
X = X[Index]#打乱数据集
y = y[Index]#打乱标签集
#分割训练集、验证集
train_percent=0.7
X_train=X[0:int(num_total * train_percent),]
X_verify=X[int(num_total * train_percent):num_total,]
y_train = y[0:int(num_total * train_percent)]
y_verify = y[int(num_total * train_percent):num_total]
#PCA 降维
pca=PCA(n_component=100)#降至 100 维
X_train_PCA = pca.fit(X_train)#利用训练集获得转化矩阵
X_verify_PCA = pca.transform(X_verify)
#数据标准化
X_train_PCA_std = Std(X_train_PCA)
X_verify_PCA_std = Std(X_verify_PCA)
'''训练'''
print("Start training:")
best_lg=None
best_acc=0
#超参数列表
LR=[1,0.5,0.1]
TOL=[0.001,0.0001]
#循环选择最优超参数
for Lr in LR:
    for Tol in TOL:
        lg = LogisticRegression(lr=Lr, tol=Tol)#创建逻辑回归模型
        lg.train(X_train_PCA_std, y_train)#读取数据进行训练
        y_pred_verify = lg.predict(X_verify_PCA_std)#利用模型预测验证集中的结果
        y_pred_verify = np.where(y_pred_verify >= 0.5, 1, 0)

acc_verify=Accuracy(y_pred_verify,y_verify)
        if acc_verify>best_acc:
            best_Lr=Lr
            best_Tol=Tol
            best_acc=acc_verify
print("最优学习率：",best_Lr)
print("最优误差阈值",best_Tol)
#测试集
np.random.seed(2020111142)
y_test=np.array(label_test)
X_test=np.array(image_test)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1] * X_test.shape[2])
Index_test = np.arange(1000)#获取序列
np.random.shuffle(Index_test)#随机打乱索引
X_test = X_test[Index_test]#打乱数据集
y_test = y_test[Index_test]#打乱标签集
#PCA 降维
pca=PCA(n_component=100)#降至 100 维
X_PCA = pca.fit(X)#利用训练集获得转化矩阵
X_test_PCA = pca.transform(X_test)

#数据标准化
X_PCA_std = Std(X_PCA)
X_test_PCA_std = Std(X_test_PCA)
'''测试集测试'''
lg_test = LogisticRegression(lr=best_Lr, tol=best_Tol)#创建逻辑回归模型
```

```
lg_test.train(X_PCA_std, y)#读取数据进行训练
y_pred = lg_test.predict(X_test_PCA_std)#利用模型预测结果
y_pre = np.where(y_pred >= 0.5, 1, 0)
ROCplot(y_test,y_pred)
acc=Accuracy(y_test,y_pre)
# KNN
class KDTree:
    def __init__(self, k_neighbors=5):
        self.k_neighbors = k_neighbors  # 保存最近邻居数 k

    def node_depth(self, i):  # 计算节点深度
        deep = np.log2(i + 1)  # 二叉树深度为 log2（总节点数+1）
        if ((deep % 1) == 0):  # 若 t 是整数则直接返回 t
            return int(deep)
        else:  # 若 t 不是整数则向上取整
            return int(deep)+1

    def sort(self, min_distance):  # 将距离降序排列
        for i in range(self.k_neighbors):
            Cur = i
            for j in range(i+1, self.k_neighbors):
                if (min_distance[j][0] > min_distance[Cur][0]):
                    Cur = j
            if (Cur != i):
                temp = min_distance[Cur]
                min_distance[Cur] = min_distance[i]
                min_distance[i] = temp

    def kd_tree_build(self, X):  # 构造 KD-Tree
        m, n = X.shape
        tree_depth = self.node_depth(m)  # 计算此时树的深度
        M = 2 ** tree_depth - 1  # 计算填满整棵树后的总节点数
        tree = np.zeros((M, 2), dtype=np.int)  # 每个节点存储一个元组: [0]实例索引, [1]切分特征索引
        tree[:, 0] = -1  # 先将所有的实例索引设置为-1，便于后期检测该节点是否是空节点
        # 创建 KD-Tree
        indices = np.arange(m)  # 用总样本数创建一个有序序列
        queue = deque([[0, 0, indices]])  # 以第 0 个节点的第 0 个特征开始构建树
        while queue:  # 每次操作清空 queue，如果最后 queue 是空的说明已经构建完毕
            # 记录树节点索引，切分特征索引和当前区域所有实例索引
            i, feature, indices = queue.popleft()
            mid = indices.size // 2  # 取第 1 个特征的中位数作为切分点
            feature_order = np.argpartition(
                X[indices, feature], mid)  # 得到将 X 按第 feature 个特征排列的索引序列
            indices = indices[feature_order]  # 按第 feature 个特征重排 indices
            tree[i, 0] = indices[mid]  # 树的每个节点第一个值为实例索引
            tree[i, 1] = feature  # 第二个值为切分特征索引
            # 若深度大于特征数，则循环使用下一特征作为切分特征
            feature = (feature + 1) % n
            # 分别将切分点左右区域的节点划分到左右子树
            li, ri = 2 * i + 1, 2 * i + 2
            if indices.size > 1:  # 若还有剩余
                queue.append([li, feature, indices[:mid]])  # 将小于部分归到左子树
            if indices.size > 2:  # 若还有多个剩余
                queue.append([ri, feature,
```

```
indices[mid+1:]])  # 将大于部分归到右子树
        return tree, tree_depth  # 返回树和树的深度


    def kd_tree_search(self, x, root, X, min_distance):  # 搜索 KD-Tree, 将最近的 k 个邻居放入大端堆
        idx = self.tree[root, 0]  # 获取根节点的索引
        if idx < 0:  # 判断根节点是否是空节点, 若是则停止搜索直接返回
            return
        root_depth = self.node_depth(root)  # 获取当前根节点深度
        i = root  # 移动到 x 所在最小超矩形区域相应的叶节点
        for _ in range(self.tree_depth - root_depth-1):  # 循环直到进入到某个叶子节点
            node = X[idx]
            l = self.tree[i, 1]  # 获取当前节点切分特征索引
            if x[l] <= node[l]:  # 根据当前节点切分特征的值, 将输入实例选择移动到左儿子或右儿子节点
                i = i * 2 + 1  # 如果小则移动到左子树寻找
            else:
                i = i * 2 + 2  # 如果大则移动到右子树寻找
            idx = self.tree[i, 0]  # 获取下一层节点的索引值
        if idx > 0:
            leaf = X[idx]
            distance = np.linalg.norm(x - leaf)  # 计算到叶节点中实例的距离
            # 进行入堆出堆操作, 更新当前 k 个最近邻居和最近距离
            if distance < min_distance[0][0]:
                min_distance[0] = (distance, idx)

                self.sort(min_distance)
        while i > root:  # 从叶子节点回退, 直到根节点停止搜索
            # 计算到父节点中实例的距离, 并更新当前最近距离
            parent_i = (i - 1) // 2  # 回退到父节点
            parent_idx = self.tree[parent_i, 0]  # 获取父节点的索引
            parent = X[parent_idx]  # 获取父节点存储的实例
            distance = np.linalg.norm(x - parent)  # 计算父节点和输入实例的距离
            # 进行入堆出堆操作, 更新当前 k 个最近邻居和最近距离
            if distance < min_distance[0][0]:
                min_distance[0] = (distance, parent_idx)
                self.sort(min_distance)
            # 获取父节点切分特征索引, 判断是否要在另一棵子树中继续搜索
            l = self.tree[parent_i, 1]
            radius = min_distance[0][0]  # 获取目前超球体半径
            # 判断超球体(x, r)是否与兄弟节点区域相交
            if np.abs(x[l] - parent[l]) < radius:  # 若相交
                # 获取兄弟节点的树索引
                if(i % 2 == 0):  # 若 i 是偶数说明叶子节点位于右子树, 兄弟节点在左子树
                    brother_i = i-1
                else:
                    brother_i = i+1
                # 递归搜索兄弟子树
                self.kd_tree_search(x, brother_i, X, min_distance)
            i = parent_i  # 向根节点回退


    def train(self, X_train, y_train):  # 训练
        # 保存训练集
        self.X_train = X_train
        self.y_train = y_train

        self.tree, self.tree_depth = self.kd_tree_build(
```

```
            X_train)  # 构造 k-d 树, 保存树及树的深度

    def predict_one(self, x):  # 对单个实例进行预测
        # 创建存储 k 个最近邻居索引的最大堆
        min_distance = []
        for i in range(self.k_neighbors):
            min_distance.append((np.inf, -1))
        # 从根开始搜索 kd tree, 将最近的 k 个邻居将存入堆
        self.kd_tree_search(x, 0, self.X_train, min_distance)
        # 获取 k 个邻居的索引
        #根据 k 个邻居中 0 或 1 出现次数决定输入实例的判断结果
        num_zero = 0
        for i in range(self.k_neighbors):
            if self.y_train[min_distance[i][1]] == 0:
                num_zero += 1
            else:
                num_zero += 0
        if num_zero > (self.k_neighbors-1)/2:
            return 0
        else:
            return 1

    def predict(self, X):  # 预测
        y_pred = []
        for i in range(X.shape[0]):
            print(i, '/', X.shape[0])
            # 对 X 中每个实例依次调用 predict_one 方法进行预测

y_pred.append(self.predict_one(X[i]))
        return(y_pred)
#随机打乱训练集
y = np.array(label)  # y 表示标签集
X = np.array(image)  # X 表示数据集
X = X.reshape(X.shape[0], X.shape[1] * X.shape[2])#展平 X 中的矩阵
Index = np.arange(num_total)#获取序列
np.random.shuffle(Index)#随机打乱索引
X = X[Index]#打乱数据集
y = y[Index]#打乱标签集
#分割训练集、验证集
train_percent=0.7
X_train=X[0:int(num_total * train_percent),]
X_verify=X[int(num_total * train_percent):num_total,]
y_train = y[0:int(num_total * train_percent)]
y_verify = y[int(num_total * train_percent):num_total]
#PCA 降维
pca=PCA(n_component=100)#降至 100 维
X_train_PCA = pca.fit(X_train)#利用训练集获得转化矩阵
X_verify_PCA = pca.transform(X_verify)
#数据标准化
X_train_PCA_std = Std(X_train_PCA)
X_verify_PCA_std = Std(X_verify_PCA)
best_acc = 0
accuracy = []
for k in range(1,21):
    knn = KDTree(k_neighbors=k)
    knn.train(X_train_PCA_std, y_train)
    y_pred = knn.predict(X_verify_PCA_std)
    err = 0
    for i in range(len(y_pred)):
        if(y_pred[i] != y_verify[i]):
            err += 1
    acc = 1-err/len(y_pred)
    accuracy.append(acc)
    if acc > best_acc:
        best_acc = acc
        best_k = k
print(best_k)
plt.plot(range(1,21),accuracy)
#测试集
y_test=np.array(label_test)
X_test=np.array(image_test)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1] * X_test.shape[2])
Index_test = np.arange(1000)#获取序列
np.random.shuffle(Index_test)#随机打乱索引
X_test = X_test[Index_test]#打乱数据集
```

```
y_test = y_test[Index_test]#打乱标签集
#PCA 降维
pca=PCA(n_component=100)#降至 100 维
X_PCA = pca.fit(X)#利用训练集获得转化矩阵
X_test_PCA = pca.transform(X_test)

#数据标准化
X_PCA_std = Std(X_PCA)
X_test_PCA_std = Std(X_test_PCA)
knn_test = KDTree(k_neighbors=best_k)
knn_test.train(X_PCA, y)
y_pred = knn_test.predict(X_test_PCA)#利用模型预测结果
ROCplot(y_test,y_pred)
acc=Accuracy(y_test,y_pred)
# SVM 对偶优化
y = np.array(label)  # y 表示标签集
X = np.array(image)  # X 表示数据集
X = X.reshape(X.shape[0], X.shape[1] * X.shape[2])#展平 X 中的矩阵
y[y==1]=-1
y[y==0]=1
print(X.shape)
length=int(len(X)/10)
print("verifysize:",length)
print("trainsize:",len(X)-length)
#划分训练集、验证集
random.seed(2020111142)
num=list(range(0,9892))
random.shuffle(num)

verify=np.zeros((10,length))
training=np.zeros((10,8903))
for i in range(1,11):
    for j in range(0,length):
        verify[i-1,j]=num[(i-1)*length+(j)]
for i in range(1,11):
    training[i-1]=np.setdiff1d(num,verify[i-1])

X_train=np.zeros((8903,3776))
X_verify=np.zeros((989,3776))
y_train=np.zeros((8903,1))
y_verify=np.zeros((989,1))
C_list=[0.1,0.5,1,5,10]
ACC=[]
accuracies=[]

#10-fold 选取 C

for C in C_list:
    for j in range(0,10):
        for i in range(0,8903):    #训练集
            X_train[i] = X[int(training[j,i])]
            y_train[i] = y[int(training[j,i])]
        for i in range(0,989):    #测试集
            X_verify[i] = X[int(verify[j,i])]
            y_verify[i] = y[int(verify[j,i])]
        #PCA 降维
        pca=PCA(n_component=100)# 降至 100 维
        X_train_PCA = pca.fit(X_train)#利用训练集获得转化矩阵
        X_verify_PCA = pca.transform(X_verify)
        #数据标准化
        X_train_PCA_std = Std(X_train_PCA)
        X_verify_PCA_std = Std(X_verify_PCA)
        #solvers.qp 二次型求解

P=matrix(np.multiply(y_train@y_train.T,X_train_PCA_std@X_train_PCA_std.T))
        q=matrix(np.ones((8903,1))*(-1))

G=matrix(np.vstack((np.identity(8903),np.identity(8903)*(-1))))

h=matrix(np.vstack((np.ones((8903,1))*C,np.zeros((8903,1)))))
        A=matrix(y_train.T)
        b=matrix(0.0)
        sol=solvers.qp(P,q,G,h,A,b)

        alpha=np.array(sol['x'])

beta_t=X_train_PCA_std.T@(alpha*y_train)
```

```
            beta_0=(1/y_train)-
X_train_PCA_std@beta_t
            #筛选求 beta0
            select=alpha.T.tolist()[0]
            index=[]
            for i in range(len(select)):
                if 0.01<select[i]<0.99:
                    index.append(i)
            beta0=np.mean(beta_0[index,])
            #预测并计算准确度

fx=X_verify_PCA_std@beta_t+beta0
            pred=np.sign(y_verify*fx)
            acc=np.sum(pred>0)
            accuracy=acc/989
            accuracies.append(accuracy)
        ACC.append(np.mean(accuracies))
print("10-fold 交叉验证准确度结果为",ACC)
print("  最  优  C  值  为
",C_list[ACC.index(max(ACC))])
print(" 最 优  C  值 对 应 的 准 确 度 为
",ACC[ACC.index(max(ACC))])
'''正式训练和测试'''
y = np.array(label)  # y 表示标签集
X = np.array(image)  # X 表示数据集
X  =  X.reshape(X.shape[0],  X.shape[1]  *
X.shape[2])#展平 X 中的矩阵
y[y==1]=-1
y[y==0]=1
y_train=np.zeros((9892,1))
y_true=np.zeros((1000,1))
for i in range(0,9892):
        y_train[i]=y[i]
for i in range(0,1000):
        y_true[i]=y_test[i]
y_true[y_true==1]=-1
y_true[y_true==0]=1
C=C_list[ACC.index(max(ACC))]
#降维
pca=PCA(n_component=100)#降至 100 维
X_PCA = pca.fit(X)#利用训练集获得转化矩
阵
X_test_PCA = pca.transform(X_test)
#数据标准化
X_PCA_std = Std(X_PCA)
X_test_PCA_std = Std(X_test_PCA)

P=matrix(np.multiply(y_train@y_train.T,X_PC
A_std@X_PCA_std.T))
q=matrix(np.ones((9892,1))*(-1))
G=matrix(np.vstack((np.identity(9892),np.ident
ity(9892)*(-1))))
h=matrix(np.vstack((np.ones((9892,1))*C,np.ze
ros((9892,1)))))
A=matrix(y_train.T)
b=matrix(0.0)
sol=solvers.qp(P,q,G,h,A,b)

#求解超平面系数 beta_t 截距 beta0
alpha=np.array(sol['x'])
beta_t=X_PCA_std.T@(alpha*y_train)
beta_0=(1/y_train)-X_PCA_std@beta_t

select=alpha.T.tolist()[0]
index=[]
for i in range(len(select)):
        if 0.01<select[i]<0.99:
            index.append(i)
#beta0=beta_0[index,][0,0]
beta0=np.mean(beta_0[index,])

#计算准确性
fx=X_test_PCA_std@beta_t+beta0
pred=np.sign(y_true*fx)

print("拟合 SVM 的系数 beta 为",beta_t)
print("拟合 SVM 的截距项 beta0 为",beta0)
def confusionSVM(y_true, y_pred):
        TP, FP, TN, FN = 0, 0, 0, 0
        for i in range(len(y_true)):
            if y_true[i] == -1 and y_pred[i] == 1:
                TP += 1
            if y_true[i] == -1 and y_pred[i] == -1:
                FN += 1
            if y_true[i] == 1 and y_pred[i] == 1:
                TN += 1
            if y_true[i] == 1 and y_pred[i] == -1:
                FP += 1
```

```
confusion_matrix=np.mat([[TP,FP],[FN,TN]])#计算预测结果的混淆矩阵
    Accuracy = (TP+TN)/(TP+TN+FN+FP)
    Recall=TP/(FN+TP)
    Precision=TP/(TP+FP)
    Specificity=TN/(TN+FP)
    F1_score = (2*Precision*Recall)/(Precision+Recall)
    print("Confusion Matrix:\n",confusion_matrix)
    print("Accuracy=",Accuracy)
    print("Recall=",Recall)
    print("Precision=",Precision)
    print("Specificity=",Specificity)
    print("F1_score=",F1_score)

ROCplot(y_true,fx)
confusionSVM(y_true,pred)

# SVM ADMM
y = np.array(label)  # y 表示标签集
X = np.array(image)  # X 表示数据集
X = X.reshape(X.shape[0], X.shape[1] * X.shape[2])#展平 X 中的矩阵
y[y==1]=-1
y[y==0]=1
print(X.shape)
length=int(len(X)/10)
print("verifysize:",length)
print("trainsize:",len(X)-length)
#划分训练集、验证集
random.seed(2020111142)
num=list(range(0,9892))
random.shuffle(num)

verify=np.zeros((10,length))
training=np.zeros((10,8903))
for i in range(1,11):
    for j in range(0,length):
        verify[i-1,j]=num[(i-1)*length+(j)]
for i in range(1,11):
    training[i-1]=np.setdiff1d(num,verify[i-1])
X_train=np.zeros((8903,3776))
X_verify=np.zeros((989,3776))
y_train=np.zeros((8903,1))
y_verify=np.zeros((989,1))
error=np.zeros((10,61))
cv=np.zeros((1,61))
lamda = np.zeros((1,61))

#10-fold 选取 lambda
for j in range(0,10):
    for i in range(0,8903):    #训练集
        X_train[i] = X[int(training[j,i])]
        y_train[i] = y[int(training[j,i])]
    for i in range(0,989):    #测试集
        X_verify[i] = X[int(verify[j,i])]
        y_verify[i] = y[int(verify[j,i])]
    #PCA 降维
    pca=PCA(n_component=100)# 降至 100 维
    X_train_PCA = pca.fit(X_train)#利用训练集获得转化矩阵
    X_verify_PCA = pca.transform(X_verify)
    #标准化
    X_train_PCA_std = Std(X_train_PCA)
    X_verify_PCA_std = Std(X_verify_PCA)

    for s in range(0,61):
        lamda[0,s] = 10**(-3+0.1*s)
        t = np.linalg.inv((X_train_PCA_std.T)@X_train_PCA_std+lamda[0,s]*np.identity((100)))@X_train_PCA_std.T
        beta_hat = t@y_train
        error[j,s] = (1/989)*np.linalg.norm(y_verify-X_verify_PCA_std@beta_hat,ord=2)\
        *np.linalg.norm(y_verify-X_verify_PCA_std@beta_hat,ord=2)

for i in range(0,61):
    cv[0,i]=np.mean(error[:,i])
s_admm=np.argmin(cv)
lamda_admm=10**(-3+0.1*s_admm)
```

```
print("10-fold 交叉验证：")
print("Selected s is {}".format(s_admm))
print("Selected lamda is {}".format(lamda_admm))
'''正式训练和测试'''
y = np.array(label)  # y 表示标签集
X = np.array(image)  # X 表示数据集
X = X.reshape(X.shape[0], X.shape[1] * X.shape[2])#展平 X 中的矩阵
y[y==1]=-1
y[y==0]=1
y_train=np.zeros((9892,1))
y_true=np.zeros((1000,1))
for i in range(0,9892):
    y_train[i]=y[i]
for i in range(0,1000):
    y_true[i]=y_test[i]
y_true[y_true==1]=-1
y_true[y_true==0]=1
#降维
pca=PCA(n_component=100)#降至 100 维
X_PCA = pca.fit(X)#利用训练集获得转化矩阵
X_test_PCA = pca.transform(X_test)
#数据标准化
X_PCA_std = Std(X_PCA)
X_test_PCA_std = Std(X_test_PCA)
rho=1
Beta1=np.zeros((100,1))
alpha1=np.zeros((9892,1))
kesi1=np.zeros((9892,1))
miu1=np.zeros((9892,1))
#ADMM 算法
for i in range(100000):
    Beta0 = Beta1
    alpha0 = alpha1
    kesi0 = kesi1
    miu0=miu1

Beta1=np.linalg.inv(lamda_admm*np.identity(100)+rho*X_PCA_std.T@X_PCA_std)\
    @(rho*X_PCA_std.T@(y_train*(1-alpha0+kesi0+miu0)))
    alpha1=np.maximum((1-y_train*X_PCA_std@Beta1+miu0-1/(9892*rho)),np.zeros((9892,1)))
    kesi1=np.maximum((-(1-y_train*X_PCA_std@Beta1)-miu0),np.zeros((9892,1)))
    miu1=miu0+1-y_train*X_PCA_std@Beta1-alpha1+kesi1
    if np.linalg.norm(Beta1-Beta0,ord=2)+np.linalg.norm(alpha1-alpha0,ord=2)\
    +np.linalg.norm(kesi1-kesi0,ord=2)+np.linalg.norm(miu1-miu0,ord=2)<0.00001:#收敛准则
        break

beta_res=Beta1
miu_res=miu1

fx=X_test_PCA_std@beta_res
pred=np.sign(y_true*fx)

print("beta 结果为",-beta_res)
ROCplot(y_true,fx)
confusionSVM(y_true,pred)
```